

Oracle7 Server™ Tuning

Release 7.3

Oracle7 Server™ Tuning

Release 7.3

Part No. A32537-1

June, 1996

ORACLE®

Oracle7 Server Tuning, Release 7.3

Part No. A32537-1

Copyright © Oracle Corporation 1995, 1996

All rights reserved. Printed in Ireland

Primary Author: John Frazzini

Contributing Author: Rita Moran

Technical Illustrator: Valarie Moore

Contributors: Jeff Cohen, Joyce Fee, Gary Hallmark, Hakan Jakobssen, Anjo Kolk, Dan Leary, Juan Loazia, Jeff Needham, Doug Rady, Ray Roccaforte, Linda Willis, Graham Wood

This software was not developed for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It is the customer's responsibility to take all appropriate measures to ensure the safe use of such applications if the programs are used for such purposes.

This software/documentation contains proprietary information of Oracle Corporation; it is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright law. Reverse engineering of the software is prohibited.

If this software/documentation is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

Restricted Rights Legend Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of DFARS 252.227-7013, Rights in Technical Data and Computer Software (October 1988).

Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

If this software/documentation is delivered to a U.S. Government Agency not within the Department of Defense, then it is delivered with "Restricted Rights", as defined in FAR 52.227-14, Rights in Data - General, including Alternate III (June 1987).

The information in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error free.

Oracle, Oracle Parallel Server, SQL*Loader, and SQL*Plus are registered trademarks of Oracle Corporation.

Oracle7, Oracle Forms, Oracle Reports, Oracle Server Manager, PL/SQL, and Pro*C are trademarks of Oracle Corporation.

All other products or company names are used for identification purposes only, and may be trademarks of their respective owners.



Preface

The Oracle7 Server is a highly tunable Relational Database Management System (RDBMS). You can enhance database performance by adjusting database applications, the database itself, and the operating system. Making such adjustments is known as *tuning*. Proper tuning of Oracle provides the best possible database performance for your specific application and hardware configuration.

This manual provides the following information:

- a step-by-step process for tuning the Oracle7 Server
- a description of diagnostic tools useful in tuning Oracle7
- installation recommendations for optimal performance
- a list of all parts of the tuning process that vary, depending on what operating system runs Oracle

Audience

This manual is an aid for those who are responsible for the operation, maintenance, and performance of an Oracle Server. To use this book, you could be a database administrator, application designer, or programmer. You should be familiar with Oracle7, the operating system, and application design before reading this manual. This manual assumes you have already read the *Oracle7 Server Concepts*, the *Oracle7 Server Application Developer's Guide*, and the *Oracle7 Server Administrator's Guide*.

How Oracle7 Server Tuning is Organized

This manual is divided into the following chapters and appendices:

Part I Overview of Tuning

Chapter 1 Overview of Tuning This chapter outlines the steps of the tuning process, describes who should be involved in tuning, and discusses how to identify performance problems.

Chapter 2 Effective Application Design This chapter describes the most popular types of applications that use Oracle and the features to examine when designing each type of system.

Part II Tuning Applications

Chapter 3 Registering Applications This chapter explains how to register applications and transactions with the database and monitor statistics by application module.

Chapter 4 SQL Processing This chapter explains the steps that Oracle must perform to process the various types of SQL commands.

Chapter 5 The Optimizer: Overview This chapter describes the optimizer, the part of Oracle that chooses the most efficient way to execute each SQL statement, and explains how the optimizer works.

Chapter 6 Parallel Query Option This chapter describes the parallel query option and parallel statement processing.

Chapter 7 Tuning SQL Statements This chapter helps you write SQL statements so they are processed more efficiently by Oracle. This chapter discusses these topics: the optimizer, indexes, clusters, hashing, and hints.

Part III Tuning the Instance

Chapter 8 Tuning Memory Allocation This chapter helps you allocate memory to Oracle data structures most efficiently. These structures include SQL and PL/SQL areas and the buffer cache.

Chapter 9 Tuning I/O This chapter discusses how to avoid I/O bottlenecks that can reduce performance. In this chapter, you learn to reduce disk contention, allocate space in data blocks, and avoid having Oracle manage space dynamically.

Chapter 10 Tuning Contention This chapter discusses the problems caused by contention and helps you recognize when these problems occur. In this chapter, you learn to reduce contention for rollback segments and redo log buffer latches.

Chapter 11 Additional Tuning Considerations This chapter discusses specific performance-related aspects of the Oracle Server, including sorts, free lists, and checkpoints.

Appendix A Performance Diagnostic Tools This appendix describes the tools that monitor performance: SQL trace, tkprof, and the EXPLAIN PLAN statement.

Appendix B Statistic Descriptions This appendix briefly describes some of the statistics that can be used for tuning.

Appendix C Managing Partition Views This appendix explains how to manage partition views.

Appendix D Operating System-Specific Information This appendix lists parts of the tuning process that vary depending on the operating system you are using. For the actual information for your operating system, see your Oracle operating system-specific documentation.

Conventions Used in this Manual

This section explains the conventions used in this manual including the following:

- icons
- text
- syntax diagrams and notation
- examples
- example data

Icons

These special icons alert you to particular information in this manual:



Suggestion: The lightbulb highlights suggestions and practical tips that could save time, make procedures easier, and so on.



Warning: The warning symbol highlights text that warns you of actions that could be particularly damaging or fatal to your system.



OSDoc

Additional Information: The OSDoc icon signifies the reader should refer to the Oracle operating system-specific documentation for additional information.

Text

This section explains the conventions used within the text:

UPPERCASE Uppercase text is used to call attention to names of SQL, PL/SQL, and Server Manager commands, SQL keywords, filenames, and initialization parameters.

italics Italicized text is used to call to attention to definitions of terms and parameters of SQL commands. Italics are also used for emphasizing certain words.

Syntax Diagrams and Notation

The syntax diagrams and notation in this manual show the syntax for SQL commands, functions, hints, and other elements. This section tells you how to read syntax diagrams and examples and write SQL statements based on them. Syntax diagrams are made up of these items:

Keywords

Keywords are words that have special meanings in the SQL language. In the syntax diagrams in this manual, keywords appear in uppercase. You must use keywords in your SQL statements exactly as they appear in the syntax diagram, except that they can be either uppercase or lowercase. For example, you must use the CREATE keyword to begin your CREATE TABLE statements just as it appears in the CREATE TABLE syntax diagram.







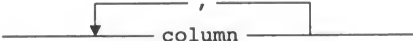
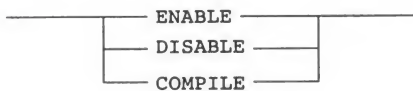
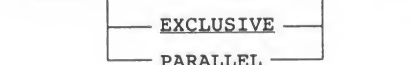
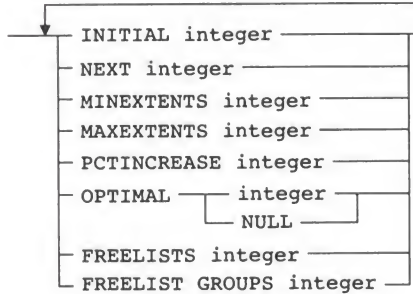
Parameters

Parameters act as place holders in syntax diagrams. They appear in lowercase. Parameters are usually names of database objects, Oracle datatype names, or expressions. When you see a parameter in a syntax diagram, substitute an object or expression of the appropriate type in your SQL statement. For example, to write a CREATE TABLE statement, use the name of the table you want to create, such as EMP, in place of the *table* parameter in the syntax diagram. Note that parameter names appear in italics in the text.

This list shows parameters that appear in the syntax diagrams in this manual and examples of the values you might substitute for them in your statements:

<i>Parameter</i>	<i>Description</i>	<i>Examples</i>
<i>table</i>	The substitution value must be the name of an object of the type specified by the parameter.	emp
<i>'text'</i>	The substitution value must be a character literal in single quotes.	'Employee records'
<i>condition</i>	The substitution value must be a condition that evaluates to TRUE or FALSE.	ename > 'A'
<i>date</i> <i>d</i>	The substitution value must be a date constant or an expression of DATE datatype.	TO_DATE('01-Jan-1994', 'DD-MON-YYYY')
<i>expr</i>	The substitution value can be an expression of any datatype.	sal + 1000
<i>integer</i>	The substitution value must be an integer.	72
<i>rowid</i>	The substitution value must be an expression of datatype ROWID.	00000462.0001.0001
<i>subquery</i>	The substitution value must be a SELECT statement contained in another SQL statement.	SELECT ename FROM emp
<i>statement_name</i> <i>block_name</i>	The substitution value must be an identifier for a SQL statement or PL/SQL block.	s1 b1

Syntax diagrams use lines and arrows to show syntactic structure. This list shows combinations of lines and arrows and their meanings within syntax diagrams:

Structure	Meaning
	The beginning of a diagram.
	The diagram continues on the next line.
	The diagram continues from the previous line.
	The end of a diagram.
	A required item (parameter or keyword). You must use this item.
	An optional item. You can use the item or omit it.
	You can optionally repeat the item multiple times. Consecutive items must be separated by a comma.
	You must use one of the items.
	You can optionally use only one of the items. If there is a default item, it is underlined.
	A list of specific items. Each item can only appear once, unless otherwise specified. The items can be listed in any order.

Your Comments Are Welcome

We value and appreciate your comments as an Oracle user and reader of the manuals. As we write, revise, and evaluate our documentation, your opinions are the most important input we receive. At the back of our printed manuals is a Reader's Comment Form, which we encourage you to use to tell us what you like and dislike about this manual or other Oracle manuals. If the form is not available, please use the following address:

Oracle7 Server Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Oracle7 Server Tuning



Contents

PART I

OVERVIEW OF TUNING

Chapter 1

- Overview of Tuning 1 – 1**
- Who Tunes? 1 – 2
- When Do You Tune? 1 – 3
- Goals for Tuning 1 – 4
- Tuning While Designing and Developing a System 1 – 5
- Tuning a Production System 1 – 6
 - Tuning the Operating System 1 – 6
 - Identifying Performance Bottlenecks 1 – 7
 - Determining the Cause of the Problem 1 – 7
 - Correcting the Problem 1 – 7
- Monitoring Facilities for Production Systems 1 – 8
 - V\$ Dynamic Performance Views 1 – 8
 - Server Manager Monitor Screens 1 – 8
 - Oracle and SNMP Support 1 – 8

Chapter 2

- Effective Application Design 2 – 1**
- Types of Applications 2 – 2
 - Online Transaction Processing (OLTP) 2 – 2
 - Decision Support 2 – 3
 - Scientific Applications 2 – 4
- Oracle Configurations 2 – 5
 - Distributed Databases 2 – 5
 - The Oracle Parallel Server 2 – 7

Parallel Query Processing	2-7
Multi-Purpose Applications	2-8
Client/Server Considerations	2-10
Minimize Network Traffic	2-10
PL/SQL	2-10

PART II

TUNING APPLICATIONS

Chapter 3

Registering Applications	3-1
Overview	3-2
Registering Applications	3-2
DBMS_APPLICATION_INFO Package	3-2
Privileges	3-3
Setting the Module Name	3-3
Example	3-3
Syntax	3-3
Setting the Action Name	3-4
Example	3-4
Syntax	3-4
Setting the Client Information	3-5
Syntax	3-5
Retrieving Application Information	3-5
Querying V\$SQLAREA	3-5
READ_MODULE Syntax	3-6
READ_CLIENT_INFO Syntax	3-6

Chapter 4

SQL Processing	4-1
SQL Statement Execution	4-2
DML Statement Processing	4-4
DDL Statement Processing	4-7
Controlling Transactions	4-7
Discrete Transactions	4-8
Deciding When to Use Discrete Transactions	4-8
How Discrete Transactions Work	4-9
Errors During Discrete Transactions	4-9
Usage Notes	4-9
Example	4-10
Serializable Transactions	4-12
Shared SQL and PL/SQL	4-13
Comparing SQL Statements and PL/SQL Blocks	4-13

Keeping Shared SQL and PL/SQL in the Shared Pool	4 – 14
The Optimizer	4 – 16
Parallel Query Option	4 – 16

Chapter 5

The Optimizer: Overview	5 – 1
What is Optimization?	5 – 2
Execution Plans	5 – 2
Oracle’s Approaches to Optimization	5 – 6
How Oracle Optimizes SQL Statements	5 – 9
Types of SQL Statements	5 – 9
Evaluating Expressions and Conditions	5 – 11
Transforming Statements	5 – 14
Optimizing Complex Statements	5 – 17
Optimizing Statements That Access Views	5 – 19
Choosing an Optimization Approach and Goal	5 – 27
Choosing Access Paths	5 – 29
Optimizing Join Statements	5 – 47
Optimizing “Star” Queries	5 – 55
Optimizing Compound Queries	5 – 58
Optimizing Distributed Statements	5 – 61
Using Histograms	5 – 62
Height-Balanced Histograms	5 – 62
When to Use Histograms	5 – 63
How to Use Histograms	5 – 63
Choosing the Number of Buckets for a Histogram	5 – 63
Viewing Histograms	5 – 64

Chapter 6

Parallel Query Option	6 – 1
Parallel Query Processing	6 – 2
Parallel Query Process Architecture	6 – 2
CREATE TABLE ... AS SELECT in Parallel	6 – 4
Parallelizing SQL Statements	6 – 5
Parallelizing Operations	6 – 6
Partitioning Rows to Each Query Server	6 – 6
Setting the Degree of Parallelism	6 – 9
Hints	6 – 10
Table and Cluster Definition Syntax	6 – 10
Default Degree of Parallelism	6 – 10
Minimum Number of Query Servers	6 – 10
Limiting the Number of Available Instances	6 – 11
Managing the Query Servers	6 – 12

Tuning for the Parallel Query Option	6 – 13
What Systems Benefit?	6 – 13
What Are the I/O Considerations?	6 – 14
How Should I Set the Degree of Parallelism?	6 – 15
Tuning the Query Servers	6 – 17
EXPLAIN PLAN	6 – 18
Parallel Index Creation	6 – 20

Chapter 7

Tuning SQL Statements	7 – 1
How to Write New SQL Statements	7 – 2
How to Use Indexes	7 – 2
How to Use Clusters	7 – 7
How to Use Hashing	7 – 8
How to Use Anti Joins	7 – 9
How to Choose an Optimization Approach	7 – 11
How to Use Hints	7 – 14
Considering Alternative Syntax	7 – 30
How to Tune Existing SQL Statements	7 – 32
Know the Application	7 – 32
Use the SQL Trace Facility	7 – 32
Tuning Individual SQL Statements	7 – 33
Bitmap Indexing	7 – 34
Benefits for Data Warehousing Applications	7 – 34
What Is a Bitmap Index?	7 – 35
Bitmap Index Example	7 – 36
When to Use Bitmap Indexing	7 – 37
How to Create a Bitmap Index	7 – 40
Initialization Parameters for Bitmap Indexing	7 – 41
Bitmap Indexes and EXPLAIN PLAN	7 – 42
Using Bitmap Access Plans on Regular B-tree Indexes	7 – 43
Restrictions	7 – 44

PART III

TUNING THE INSTANCE

Chapter 8

Tuning Memory Allocation	8 – 1
The Importance of Memory Allocation	8 – 2
Steps for Tuning Memory Allocation	8 – 2
Tuning Your Operating System	8 – 2
Tuning Private SQL and PL/SQL Areas	8 – 2
Tuning the Shared Pool	8 – 3

Tuning the Buffer Cache	8-3
Tuning Your Operating System	8-3
Reducing Paging and Swapping	8-4
Tuning the System Global Area (SGA)	8-4
User Memory Allocation	8-5
Tuning Private SQL and PL/SQL Areas	8-6
Tuning the Shared Pool	8-8
Tuning the Library Cache	8-8
Tuning the Data Dictionary Cache	8-14
Tuning the Shared Pool with Multi-Threaded Server	8-16
Tuning the Buffer Cache	8-18
Examining Buffer Cache Activity	8-18
Reducing Buffer Cache Misses	8-19
Removing Unnecessary Buffers	8-21
Reallocating Memory	8-24

Chapter 9

Tuning I/O	9-1
The Importance of Tuning I/O	9-2
Reducing Disk Contention	9-2
What Is Disk Contention?	9-2
Monitoring Disk Activity	9-2
Distributing I/O	9-4
Allocating Space in Data Blocks	9-8
Migrated and Chained Rows	9-8
Avoiding Dynamic Space Management	9-10
Detecting Dynamic Extension	9-10
Allocating Extents	9-11
Avoiding Dynamic Space Management in Rollback Segments	9-11

Chapter 10

Tuning Contention	10-1
Reducing Contention for Rollback Segments	10-2
Identifying Rollback Segment Contention	10-2
Creating Rollback Segments	10-3
Reducing Contention for Multi-Threaded Server Processes ...	10-4
Reducing Contention for Dispatcher Processes	10-4
Reducing Contention for Shared Server Processes	10-6
Reducing Contention for Query Servers	10-8
Identifying Query Server Contention	10-8
Reducing Query Server Contention	10-8
Reducing Contention for Redo Log Buffer Latches	10-9

Space in the Redo Log Buffer	10 – 9
Redo Log Buffer Latches	10 – 9
Examining Redo Log Activity	10 – 10
Reducing Latch Contention	10 – 12
Reducing LRU Latch Contention	10 – 13
Assigning All Oracle Processes Equal Priority	10 – 14

Chapter 11

Additional Tuning Considerations	11 – 1
Tuning Sorts	11 – 2
Allocating Memory for Sort Areas	11 – 2
Optimizing Sort Performance with TEMPORARY Tablespaces	11 – 3
Avoiding Sorts	11 – 3
GROUP BY NOSORT	11 – 4
SORT_DIRECT_WRITES parameter	11 – 5
Recreating an Index	11 – 6
Reducing Free List Contention	11 – 7
Identifying Free List Contention	11 – 7
Adding More Free Lists	11 – 8
Tuning Checkpoints	11 – 8
How Checkpoints Affect Performance	11 – 8
Choosing Checkpoint Frequency	11 – 9
Reducing the Performance Impact of a Checkpoint	11 – 9

PART IV

REFERENCE

Appendix A

Performance Diagnostic Tools	A – 1
Dynamic Performance Views	A – 2
Identifying Performance Bottlenecks	A – 2
Determining the Cause of the Problem	A – 5
The SQL Trace Facility	A – 7
Using the SQL Trace Facility	A – 7
Setting Initialization Parameters for SQL Trace Facility ...	A – 8
Enabling the SQL Trace Facility	A – 8
Running TKPROF	A – 10
Interpreting TKPROF Output	A – 15
Storing SQL Trace Facility Statistics	A – 18
The EXPLAIN PLAN Command	A – 21
Creating the Output Table	A – 22
Output Table Columns	A – 23
Example of EXPLAIN PLAN Output	A – 27

Oracle Trace	A – 30
Instrumentation of Oracle For Oracle Trace	A – 30
Enabling and Disabling Oracle Trace	A – 30

Appendix B

Statistic Descriptions	B – 1
buffer busy waits	B – 2
consistent changes	B – 2
consistent gets	B – 2
db block changes	B – 2
db block gets	B – 2
free buffer waits	B – 3
parse count	B – 3
physical reads	B – 3
physical writes	B – 3
recursive calls	B – 3
redo entries	B – 3
redo log space requests	B – 4
redo sync writes	B – 4
sorts (disk)	B – 4
sorts (memory)	B – 4
table fetch rowid	B – 4
table fetch continued row	B – 5
table scan blocks	B – 5
table scan rows	B – 5
table scans (long tables)	B – 5
table scans (short tables)	B – 5
user calls	B – 6
user commits	B – 6
user rollbacks	B – 6
write requests	B – 6

Appendix C

Managing Partition Views	C – 1
Partition View Guidelines	C – 2
Partition View Highlights	C – 2
Rules and Guidelines for Use	C – 2
Defining Partition Views	C – 4
Defining Partition Views Using Check Constraints	C – 4
Defining Partition Views Using WHERE Clauses	C – 5
Partition Views: Example	C – 5
Create the Tables Underlying the Partition View	C – 6
Load Each Partition	C – 6

Enable Check Constraints	C – 6
Add Additional Overlapping Partition Criteria	C – 6
Create Indexes for Each Partition	C – 7
Analyze the Partitions	C – 7
Create the View that Ties the Partitions Together	C – 8
Partition Views and Parallelism	C – 8

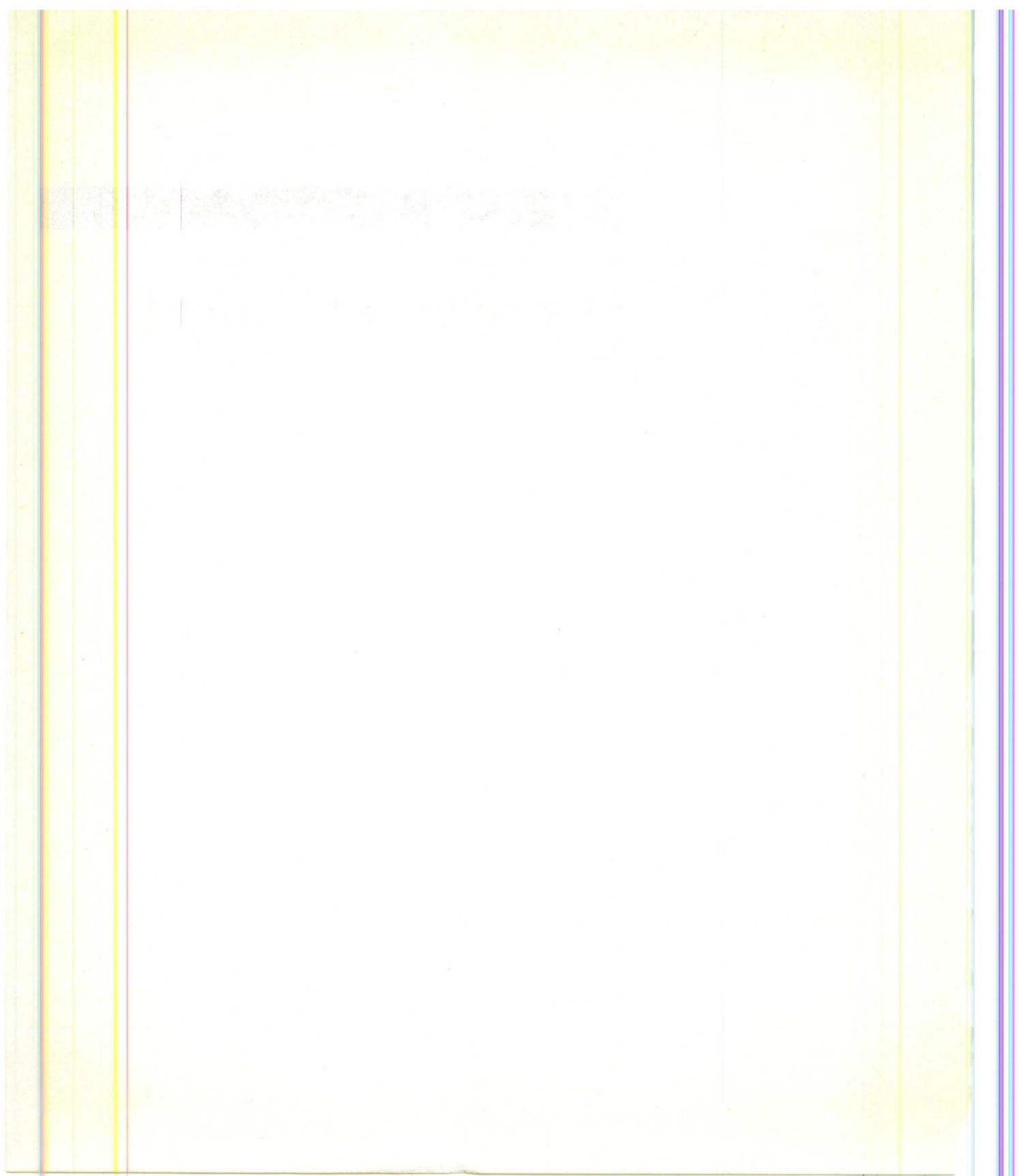
Appendix D

Operating System–Specific Information	D – 1
File Striping, 6 – 14	D – 2
I/O Monitoring and Tuning, 9 – 2	D – 2
Memory Tuning, 8 – 3	D – 2
Monitoring CPU, 6 – 15	D – 2
Tuning Your Operating System, 1 – 6	D – 2
Parallel Query Tuning, 6 – 13	D – 2

PART

I

Overview of Tuning



Overview of Tuning

The Oracle Server is a sophisticated and highly tunable software product. This chapter describes the tuning process and the people who should be involved in tuning an Oracle Server and its associated operating-system hardware and software. Topics in this chapter include

- who tunes the system?
- when do you tune the system?
- setting goals for effective tuning
- tuning while designing and developing a system
- tuning a production system
- monitoring a production system

Who Tunes?

Several people must exchange information and become involved in the tuning process to tune the system effectively. For example:

- The application designer must communicate the system design so that everyone can understand the flow of data in an application.
- The application developers must communicate the implementation strategies they choose so that modules and SQL statements can be quickly and easily identified during statement tuning.
- The database administrator must carefully monitor and document system activity so that unusual system performance can be identified and corrected.
- The hardware/software administrators must document and communicate the hardware software configuration of the system so that everyone can design and administer the system effectively.

In short, everyone involved with the system has some role in the tuning process. When the people mentioned above communicate and document the system's characteristics, tuning becomes significantly easier and faster.

In practice, unfortunately, the database administrator usually has the primary responsibility for tuning. Also, the database administrator rarely has adequate documentation of the system. The section "Tuning a Production System" on page 1 – 6 describes how database administrators can identify performance problems given little or no information from the application designers.

When Do You Tune?

Most people believe the tuning process begins when users complain about poor response time. This is usually too late in the process to use some of the most effective tuning strategies. At that point, if you are unwilling to completely redesign the application, you may only improve performance marginally by reallocating memory and tuning I/O. Oracle provides many features that can greatly improve performance when properly used in a well designed system.

The application designer needs to set performance expectations of the application at the design phase. Then during design and development, the application designers should consider which Oracle features can benefit the system. This manual describes some of the performance implications of the various Oracle features, and you can use this information to determine if a particular feature will enhance the performance of your application.

By designing a system to perform well, you can eliminate cost and frustration later in the life of the application. Figure 1 – 1 and Figure 1 – 2 illustrate the relative cost and benefits of tuning during the life of an application. As you can see, the most effective time to tune is during the design phase. Tuning during design gives you the maximum benefit at the lowest cost.

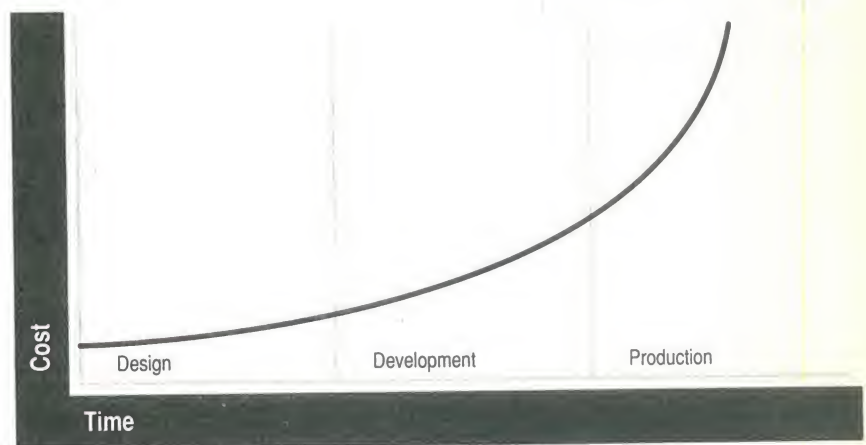


Figure 1 – 1 Cost of Tuning During the Life of an Application

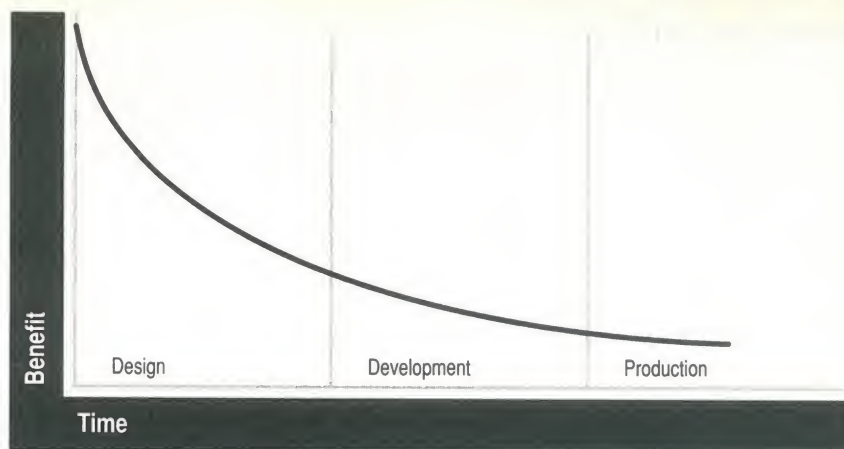


Figure 1 – 2 Benefit of Tuning During the Life of an Application

Chapter 2, “Effective Application Design”, describes some of the performance enhancing features you can use to ensure your application meets its designed performance goals. Of course, even the performance of well designed systems can degrade with use. This manual also describes a strategy for identifying and dealing with various performance bottlenecks.

Goals for Tuning

Whether you are designing or maintaining a system, you should set specific performance goals so that you know when to tune. You can needlessly spend time tuning your system without significant gain if you attempt to haphazardly alter initialization parameters or SQL statements. The most effective method for tuning your system is the following:

1. Consider performance when designing the system.
2. Tune the operating-system hardware and software.
3. Identify performance bottlenecks.
4. Determine the cause of the problem.
5. Take corrective action.

When you are designing your system, set a specific goal; for example, a response time of less than three seconds. When the application does not meet that goal, identify the bottleneck causing the slowdown (for example, I/O contention), determine the cause, and take corrective action. During development, you should test the application to determine if it meets the designed performance goals before deploying the application.

When you are maintaining a production system, there is a quick and effective way to identify performance bottlenecks. This method is described in the section “Tuning a Production System” on page 1 – 6.

In any event, tuning is usually a series of tradeoffs. Once you have determined the bottlenecks, you may have to sacrifice some other areas to achieve the desired results. For example, if I/O is a problem, you may need to purchase more memory or more disks. If a purchase is not possible, you may have to limit the concurrency of the system to achieve the desired performance. However, if you have clearly defined goals for performance, the decision on what to trade for higher performance is simpler because you have stated which areas are the most important.

Tuning While Designing and Developing a System

Well designed systems can prevent performance problems later in the life of an application. System designers and application developers must understand Oracle’s query processing mechanism to write effective SQL statements. Chapter 2, “Effective Application Design”, discusses various configurations for your system and which applications are best suited for each configuration. Chapter 5, “The Optimizer”, discusses Oracle’s query optimizer and how to write statements that achieve the fastest results.

When designing your system, use the following guidelines for optimal performance:

- Eliminate unnecessary network traffic in client/server applications.
- Use the appropriate Oracle Server options (for example, parallel query or distributed database) for your system.
- Use default Oracle locking unless your application has specific needs.
- Register application modules with the database so that you can track performance on a per-module basis.

- Choose the best size for your data blocks.
- Distribute your data so that the data a node uses is stored locally on that node.

Chapter 2 and Chapter 5 discuss database design issues more completely. Refer to those chapters before building your applications.

Tuning a Production System

This section describes a method for quickly and easily finding performance bottlenecks and determining the corrective action for a production system. This method relies on a firm understanding of Oracle Server architecture and features. You should be familiar with the content of *Oracle7 Server Concepts* before attempting to tune your system.

Follow these steps to tune your existing system:

1. Tune the operating-system hardware and software.
2. Identify performance bottlenecks by querying the V\$SESSION_WAIT view. This dynamic performance view lists events that cause sessions to wait.
3. Determine the cause of the bottleneck by analyzing the data in V\$SESSION_WAIT.
4. Correct the problem.

Tuning the Operating System

Before you can effectively tune Oracle, you must ensure that the operating system is at its peak performance. You must work closely with the hardware/software system administrators to ensure that Oracle is allocated the proper operating-system resources.

Subsequent chapters of this manual describe some memory and I/O operating system issues. Unfortunately, this manual cannot cover every operating system-specific topic.



OSDoc

Additional Information: Tuning your operating system is different for every operating system Oracle runs on. Refer to your operating-system hardware/software documentation as well as your Oracle operating system-specific documentation for more information on tuning your operating system.

**Identifying
Performance
Bottlenecks**

The V\$SESSION_WAIT view lists the events that cause all user and system sessions to wait. Querying this view is a good way to get a quick look at possible performance bottlenecks. This view lists what each session is currently waiting for or what the session last waited for. It shows wait time in hundredths of a second for each session wait event.

**Determining the Cause
of the Problem**

Each performance problem has a unique cause, and you may need to query several dynamic performance views to find the cause of some problems. This manual groups related performance problems into chapters. The table in the previous section points you to the proper chapter for each type of problem. Performance problems tend to fall into one of these categories:

- CPU
- memory
- I/O
- contention for latches or other structures

The V\$SESSION_WAIT table helps you to categorize the performance problem and place it into one of these three categories. Also, there is additional information in columns P1, P2, and P3 for some events in V\$SESSION_WAIT. Once you have queried V\$SESSION_WAIT to get a general idea of the type of performance problem, refer to the appropriate chapter in this manual to diagnose the problem specifically.

For more information about using dynamic performance views to identify bottlenecks and determine the cause of the problem, see “Dynamic Performance Views” on page A – 2.

Correcting the Problem

Most performance problem have a unique corrective solution. The chapters of this manual explain how to correct most types of performance problems. Once you have determined the type of problem with V\$SESSION_WAIT, read the appropriate chapter of this manual, and then you can take corrective action.

In the example, the redo allocation and redo copy latches are suffering from contention. To correct the redo allocation latch problem, you must decrease the value of the initialization parameter LOG_SMALL_ENTRY_MAX_SIZE. To correct the redo copy latch problem, you must increase the value of the initialization parameter LOG_SIMULTANEOUS_COPIES.

Each chapter in this manual suggests corrective action for the various performance problems you may encounter. You must read the chapter concerning your problem to properly diagnose and solve the performance problem.

Monitoring Facilities for Production Systems

There are several facilities available for monitoring production systems and determining performance problems. This manual uses the V\$ views that Oracle provides to illustrate performance problems. There are also graphic monitors provided by Server Manager that contain the same information. Oracle also supports the Simple Network Management Protocol (SNMP) through two MIBs (management information base) of database information.

Because the V\$ views are available to all database users, this book uses only V\$ views in the examples of gathering statistics.

V\$ Dynamic Performance Views

This manual demonstrates the use of V\$ dynamic performance views that Oracle provides to monitor your system. The database user SYS owns these views, and administrators can grant access to these views to any database user. Only some of these views are relevant to tuning your system. For a complete description of all V\$ dynamic performance views, see the *Oracle7 Server Reference*.

Server Manager Monitor Screens

Oracle Server Manager has several monitor screens that you can use to monitor database activity. All of these monitors are based on the V\$ dynamic performance views. See the *Oracle Server Manager User's Guide* for a description of all the monitors and how they relate to the V\$ views.

Oracle and SNMP Support

SNMP is acknowledged as the standard, open protocol for heterogeneous management applications. Oracle SNMP support enables Oracle databases to be discovered on the network, identified, and monitored by any SNMP-based management application. Oracle supports two database MIBs: the proposed standard MIB for RDBMSs (independent of vendor), and an Oracle-specific MIB, which contains Oracle-specific information.

Some statistics mentioned in this manual are supported by these two MIBs, and others are not. If a statistic can be obtained through SNMP, it will be noted when that statistic is mentioned in this manual. For more information about Oracle's SNMP support, see the *Oracle SNMP Support Reference Guide*.

Effective Application Design

This chapter describes the various types of applications that use Oracle databases and the suggested approaches and features available when designing each. Topics in this chapter include

- types of applications
- Oracle configurations
- client–server design considerations

Types of Applications

There are thousands of types of applications that you can build on top of an Oracle Server. This section attempts to categorize the most popular types of applications and describe the design considerations for each. Each section lists topics that are crucial for performance for that type of system.

Refer to the *Oracle7 Server Concepts* manual, the *Oracle7 Server Application Developer's Guide*, and the *Oracle7 Server Administrator's Guide* for more information on these topics and how to implement them in your system.

Online Transaction Processing (OLTP)

Online transaction processing (OLTP) applications are high-throughput, insert/update intensive systems. These systems are characterized by constantly growing large volumes of data that several hundred users access concurrently. Typical OLTP applications are airline reservation systems, large order-entry applications, and banking applications. The key goals of an OLTP system are availability, speed, concurrency, and recoverability.

Figure 2 – 1 illustrates the interaction between an OLTP application and an Oracle Server.

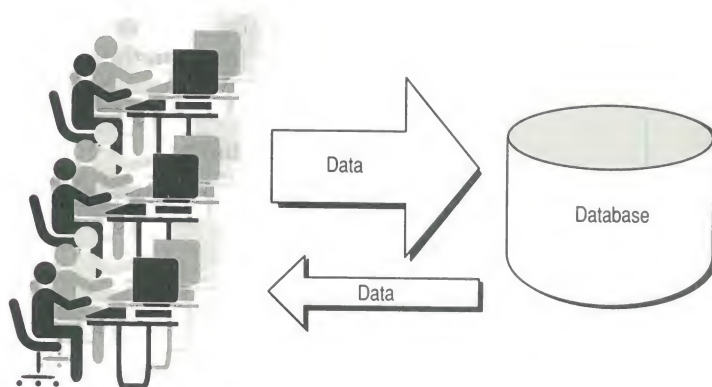


Figure 2 – 1 Online Transaction Processing Systems

When you design an OLTP system, you must ensure that the large number of concurrent users does not interfere with the system's performance. You must also avoid excessive use of indexes and clusters because these structures slow down insert and update activity. The following topics are crucial in tuning an OLTP system:

- rollback segments
- indexes, clusters, and hashing
- discrete transactions

- data block size
- dynamic allocation of space to tables and rollback segments
- transaction processing monitors and the multi-threaded server
- the shared pool
- well tuned SQL statements
- integrity constraints
- client-server architecture
- procedures, packages, and functions

Refer to the *Oracle7 Server Concepts* manual and the *Oracle7 Server Administrator's Guide* for a description of each of these topics. Read about these topics before designing your system and decide which features can benefit your particular situation.

Decision Support

Decision support applications distill large amounts of information into understandable reports. Typically, decision support applications perform queries on the large amount of data gathered from OLTP applications. Decision makers in an organization use these applications to determine what strategies the organization should take based on the available information.

Figure 2 – 2 illustrates the interaction between a decision support application and an Oracle Server.

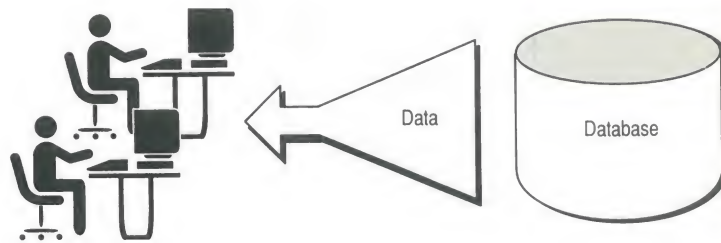


Figure 2 – 2 Decision Support Systems

An example of a decision support system is a marketing tool that determines the buying patterns of consumers based on information gathered from demographic studies. The demographic data is gathered and entered into the system, and the marketing staff queries this data to determine which items sell best in which locations. This report helps to decide which items to purchase and market in the various locations.

The key goals of a decision support system are speed, accuracy, and availability.

When you design a decision support system, you must ensure that queries on large amounts of data can perform within a reasonable time frame. Decision makers often need reports on a daily basis, so you may need to guarantee that the report can complete overnight. The key to performance in a decision support system is properly tuned queries and proper use of indexes, clusters, and hashing. The parallel query option can also benefit decision support systems. The following topics are crucial in tuning a decision support system:

- indexes, clusters, hashing
- data block size
- parallel query option
- the optimizer
- using hints in queries
- PL/SQL functions in SQL statements

Scientific Applications Scientific or statistical systems often perform complex calculations on large amounts of data. Often the data represents complicated datatypes, like latitude or pressure. These applications are characterized by diverse datatypes and sophisticated SQL calculations used to produce summary reports.

Figure 2 – 3 illustrates the interaction between a scientific application and an Oracle Server.

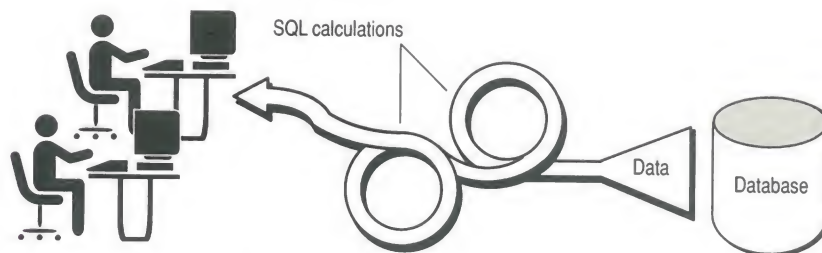


Figure 2 – 3 Scientific Applications

An example of a scientific or statistical system is a meteorological application that gathers data from weather satellites. The application performs complicated statistical analysis of the data to predict future weather patterns.

The key goals of scientific applications are accuracy and speed.

When you design a scientific or statistical system, you must ensure that complicated queries are not consuming all of the system resources.

Properly tuning queries is the primary concern in these systems. The following topics are crucial in tuning a scientific or statistical system:

- PL/SQL functions in SQL statements
- parallel query option
- the optimizer
- using hints in queries

Oracle Configurations

You can configure your system depending on the hardware and software available to you. The basic configurations are distributed databases, Oracle Parallel Server, the parallel query option, or client/server. Depending on your application and your operating system, each of these or a combination of these configurations will best suit your needs.

The following sections briefly describe distributed databases, the Oracle Parallel Server, and the parallel query option. See "Client/Server Considerations" on page 2 – 10 for more information about the client/server configuration.

Distributed Databases

Distributed database systems involve distributing data over multiple databases on multiple machines. Several smaller server machines can be cheaper and more flexible than one large, centrally located server. This configuration becomes more popular with the advent of small, powerful server machines and cheaper connectivity options. Distributed systems allow you to have data physically located at several sites, and each site can transparently access all of the data.

Figure 2 – 4 illustrates the distributed database configuration of the Oracle Server.

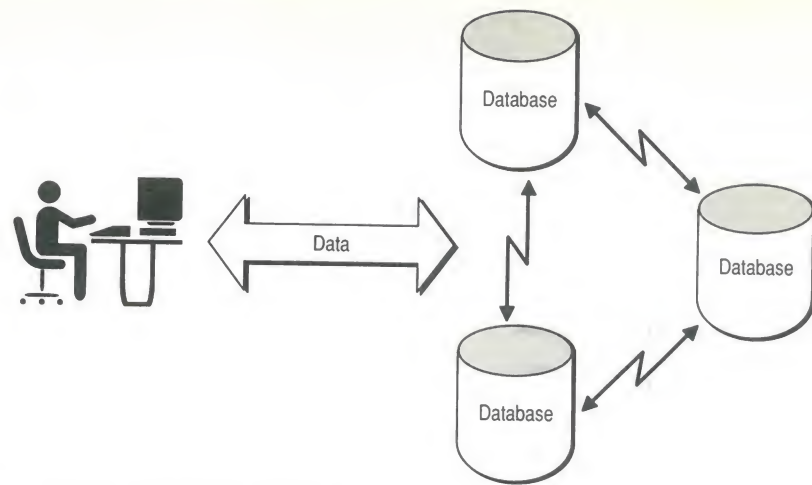


Figure 2 – 4 Distributed Databases

An example of a distributed database system is a mail order application with order entry clerks in several locations across the country. Each clerk has access to a copy of the central inventory database, but the clerks perform local operations on an order-entry system. The local orders are forwarded each day to the central shipping department. While the inventory and shipping departments are centrally located, the clerks are spread across the country for the convenience of the customers.

The key goals of a distributed database system are availability, accuracy, concurrency, and recoverability. Refer to *Oracle7 Server Distributed Systems, Volume I* and *Oracle7 Server Distributed Systems, Volume II* for more information on distributed databases.

When you design a distributed database system, the location of the data is the most important factor. You must ensure that local clients have quick access to the data they use most frequently and remote operations do not occur very often. The following topics are crucial to the design of distributed database systems:

- SQL*Net
- distributed database design
- symmetric replication
- table snapshots and snapshot logs
- procedures, packages, and functions

Refer to *Oracle7 Server Distributed Systems, Volume I* and *Oracle7 Server Distributed Systems, Volume II* for more information on distributed databases and the distributed option.

The Oracle Parallel Server

The Oracle Parallel Server is available on clustered or massively parallel systems that support shared disks. A Parallel Server allows multiple machines to have separate instances all accessing the same database. This configuration greatly enhances data throughput.

Figure 2 – 5 illustrates the Parallel Server option of the Oracle Server.



Figure 2 – 5 An Oracle Parallel Server

When you configure a system with the Parallel Server option, your key concern is data contention among the various nodes. Each node that requires data must first obtain a lock on that data to ensure data consistency. If multiple nodes all want to access the same data, that data must first be written to disk, and then the next node can obtain the lock. This type of contention can significantly slow a Parallel Server, so a Parallel Server must effectively partition the data among the various nodes.

Refer to the *Oracle7 Parallel Server Concepts & Administration* manual for more information on the Parallel Server option.

Parallel Query Processing

With the parallel query option, multiple processes can work together simultaneously to process a single SQL statement. This capability is called *parallel query processing*. By dividing the work necessary to process a statement among multiple server processes, the Oracle Server can process the statement more quickly than if only a single server process processed it.

Figure 2 – 6 illustrates the parallel query option of the Oracle Server.

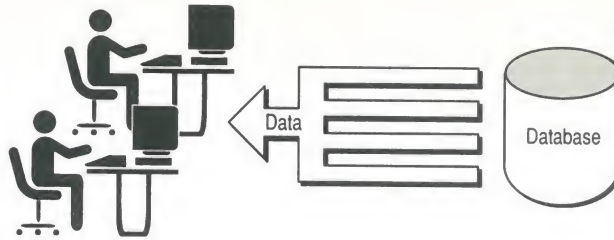


Figure 2 – 6 Parallel Query Processing

The parallel query option can dramatically improve performance for data-intensive operations associated with decision support applications or very large database environments. Symmetric multiprocessing (SMP), clustered, or massively parallel systems gain the largest performance benefits from the parallel query option because query processing can be effectively split among many CPUs on a single system.

The parallel query option helps systems scale in performance when adding hardware resources. If your system's CPUs and disk controllers are already heavily loaded, you need to alleviate the system's load before using the parallel query option to improve performance. The section "Tuning for the Parallel Query Option" on page 6 – 13 describes how your system can achieve the best performance with the parallel query option.

Refer to Chapter 6, "Parallel Query Option", for more information about the parallel query option and parallel query processing.

Multi-Purpose Applications

Many applications rely on several of the configurations and options mentioned in this section. You must decide what type of activity your application performs and determine which features are best suited for it. One typical multi-purpose configuration is a combination of OLTP and decision support systems. Often data gathered by an OLTP application "feeds" a decision support system.

Figure 2 – 7 illustrates multiple configurations and applications accessing an Oracle Server.

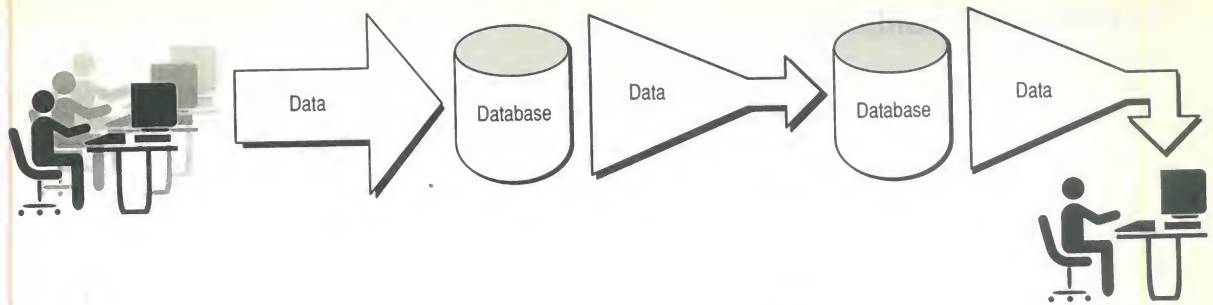


Figure 2 – 7 A Hybrid OLTP/Decision Support System

An example of a combination OLTP/decision support system is a marketing tool that determines the buying patterns of consumers based on information gathered from retail stores. The retail stores gather a large amount of data from daily purchases, and the marketing staff queries this data to determine which items sell best in which locations. This report is then used to decide what items to stock more of in inventory in each store.

In this example, both systems could use the same database, but the conflicting goals of OLTP and decision support cause performance problems for both parts of the system. To solve this problem, an OLTP database stores the data gathered by the retail stores, then an image of that data is copied into a second database, which is queried by the decision support application. This configuration slightly compromises the goal of accuracy for the decision support application (the data is only copied once per day), but the tradeoff is significantly better performance from both systems.

For hybrid systems, refer to the features mentioned in each of the previous sections, then determine which goals are most important. You will most probably need to compromise one or more of these goals to achieve acceptable performance across the whole system.

Client/Server Considerations

Client/server architecture distributes the work of a system between the client (application) machine and the server (in this case an Oracle Server). Typically, client machines are workstations that execute a graphical user interface (GUI) application that connects to a larger server machine that houses the Oracle Server.

Because the user typically interacts more heavily with the application, response time is maximized by executing the client on the user's local workstation. In this environment, you should keep communication with the server to a minimum because requests to the server must travel over a network. Too much network traffic can significantly decrease a client application's performance.

Minimize Network Traffic

The Oracle Server provides several ways to minimize the communication between the client application and the database server. Generally, your goal is to group several related server requests (SQL statements) into one request over the network. The following Oracle Server features can reduce network traffic by grouping related statements, or performing several client functions with one network call:

- stored procedures
- database triggers
- integrity constraints
- array processing
- sequences

Refer to *Oracle7 Server Concepts* for more information about these features and how they can reduce network traffic. When possible, use these features when designing your client/server applications. Most of the features above use PL/SQL, Oracle's procedural extension to SQL. The next section describes how PL/SQL can minimize your network traffic.

PL/SQL

Anonymous blocks and stored procedures improve performance by reducing calls from your application to Oracle. Reducing calls is especially helpful in networked environments where calls may incur a large overhead. PL/SQL reduces calls in these ways:

- Your applications can pass multiple SQL statements to Oracle at once.
- Your Oracle Forms applications can perform procedural operations without calling Oracle.

Stored procedures further improve performance by eliminating parsing and automatically taking advantage of shared PL/SQL areas.

Packages further improve performance in these ways:

- Storing related procedures and functions together in a package reduces the amount of I/O necessary to read them into memory.
- Storing a procedure in a package allows you to redefine the procedure without causing Oracle to recompile other procedures that call it.

The PL/SQL Engine in the Oracle Server

Anonymous blocks and stored procedures allow applications to pass multiple SQL statements to Oracle at once. A block containing multiple SQL statements can be passed to Oracle in a single call. Without PL/SQL, applications must pass SQL statements one at a time, each with a separate call to Oracle. Figure 2 – 8 illustrates how PL/SQL reduces network traffic and boosts performance.

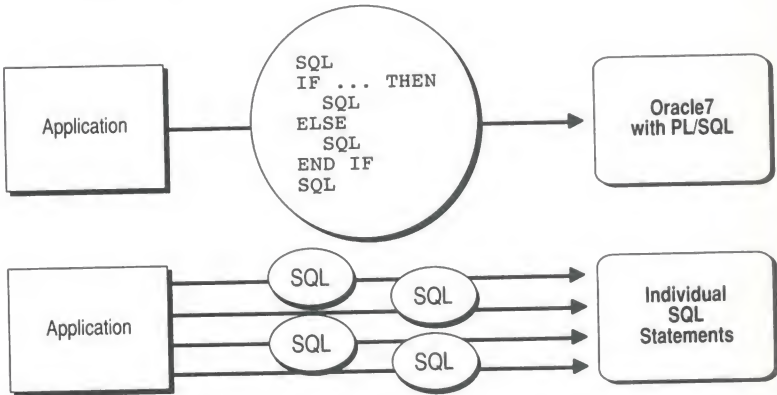


Figure 2 – 8 PL/SQL Boosts Performance

The PL/SQL Engine in Oracle Application Tools

Anonymous blocks allow Oracle Forms applications to perform procedural functions that might otherwise require calls to Oracle. Since a PL/SQL block can contain procedural statements, your application can use the PL/SQL engine incorporated into the Oracle application tool to execute such statements.

For example, you should perform data calculations in your Oracle Forms applications by passing procedural statements to the PL/SQL engine, rather than by issuing SQL statements. SQL statements require calls to Oracle, while the procedural statements can be processed by the PL/SQL engine in Oracle Forms itself. In this case, PL/SQL helps you avoid calling Oracle altogether.

Using PL/SQL in Your Applications

You should use PL/SQL for parts of your applications that perform a great deal of database access. Follow these guidelines for using PL/SQL with specific Oracle application tools to improve the performance of your applications:

Oracle Precompilers	For database-intensive embedded SQL programs, pass multiple SQL statements to Oracle by issuing anonymous blocks or calling stored procedures rather than by issuing multiple embedded SQL statements.
SQL*Plus	For small procedural tasks, issue anonymous blocks or call stored procedures from SQL*Plus rather than using Oracle Reports or Oracle Precompiler programs.
Oracle Forms	Replace multiple trigger steps in Oracle Forms applications with single trigger steps containing anonymous blocks.

PART

II

Tuning Applications

Registering Applications

Application developers can record the name of the executing module or transaction in the database for use later when tracking the performance of various modules. This chapter describes how to register an application with the database and retrieve statistics on each registered module or code segment. Topics in this chapter include:

- an overview
- registering applications
- setting the module name
- setting the action name
- setting the client info
- retrieving the application information

Overview

Oracle provides a method for applications to register the name of the application and actions performed by that application with the database. Registering the application allows system administrators and performance tuning specialists to track performance by module. System administrators can also use this information to track resource usage by module. When an application registers with the database, its name and actions are recorded in the V\$SESSION and V\$SQLAREA views.

Your applications should set the name of the module and name of the action automatically each time a user enters that module. The module name could be the name of a form in an Oracle Forms application, or the name of the code segment in an Oracle Precompilers application. The action name should usually be the name or description of the current transaction within a module.

Registering Applications

To register applications with the database, use the procedures in the DBMS_APPLICATION_INFO package.

DBMS_APPLICATION_INFO Package

Table 3 – 1 lists the procedures available in the DBMS_APPLICATION_INFO package.

Procedure	Description	Discussed on
SET_MODULE	Sets the name of the module that is currently running.	page 3 – 3
SET_ACTION	Sets the name of the current action within the current module.	page 3 – 4
SET_CLIENT_INFO	Sets the client information field for the session.	page 3 – 5
READ_MODULE	Reads the values of the module and action fields for the current session.	page 3 – 6
READ_CLIENT_INFO	Reads the client information field for the current session.	page 3 – 6

Table 3 – 1 Procedures in the DBMS_APPLICATION_INFO Package

Privileges

Before using this package, you must run the DBMSUTL.SQL script to create the DBMS_APPLICATION_INFO package. For more information about Oracle supplied packages and executing stored procedures, see *Oracle7 Server Application Developer's Guide*.

Caution: Do not use procedures in the DBMS_APPLICATION_INFO package if you are using the Trusted Oracle7 Server. For more information about the Trusted Oracle7 Server, see the *Trusted Oracle7 Server Administrator's Guide*.

Setting the Module Name

To set the name of the current application or module, use the SET_MODULE procedure in the DBMS_APPLICATION_INFO package. The module name should be the name of the procedure (if using stored procedures), or the name of the application. The action name should be descriptive text about the action the module is performing.

Example

The following is an example of a PL/SQL block that sets the module name and action name:

```
CREATE PROCEDURE add_employee(
    name          VARCHAR2(20),
    salary         NUMBER(7,2),
    manager        NUMBER,
    title          VARCHAR2(9),
    commission     NUMBER(7,2),
    department     NUMBER(2)) AS
BEGIN
    DBMS_APPLICATION_INFO.SET_MODULE(
        module_name => 'add_employee',
        action_name => 'insert into emp');
    INSERT INTO emp
        (ename, empno, sal, mgr, job, hiredate, comm, deptno)
        VALUES (name, next.emp_seq, manager, title, SYSDATE,
            commission, department);
    DBMS_APPLICATION_INFO.SET_MODULE('','');
END;
```

Syntax

The parameters for the SET_MODULE procedure are described in Table 3 – 2. The syntax for this procedure is shown below:

```
DBMS_APPLICATION_INFO.SET_MODULE(
    module_name  IN    VARCHAR2,
    action_name  IN    VARCHAR2)
```

Parameter	Description
module_name	The name of the module that is currently running. When the current module terminates, call this procedure with the name of the new module if there is one, or null if there is not. Names longer than 48 bytes are truncated.
action_name	The name of the current action within the current module. If you do not want to specify an action, this value should be null. Names longer than 32 bytes are truncated.

Table 3 – 2 Parameters for SET_MODULE Procedure

Setting the Action Name

To set the name of the current action within the current module, use the SET_ACTION command in the DBMS_APPLICATION_INFO package. The action name should be descriptive text about the current action being performed. You should probably set the action name before the start of every transaction.

Example

The following is an example of a transaction that uses the registration procedure:

```
CREATE OR REPLACE PROCEDURE bal_tran (amt IN NUMBER(7,2)) AS
BEGIN
  -- balance transfer transaction
  DBMS_APPLICATION_INFO.SET_ACTION(
    action_name => 'transfer from chk to sav');
  UPDATE chk SET bal = bal + :amt
    WHERE acct# = :acct;
  UPDATE sav SET bal = bal - :amt
    WHERE acct# = :acct;
  COMMIT;
  DBMS_APPLICATION_INFO.SET_ACTION('');
END;
```

Set the transaction name to null after the transaction completes so that subsequent transactions are logged correctly. If you do not set the transaction name to null, subsequent transactions may be logged with the previous transaction's name.

Syntax

The parameter for the SET_ACTION procedure is described in Table 3 – 3. The syntax for this procedure is shown below:

```
DBMS_APPLICATION_INFO.SET_ACTION(action_name IN VARCHAR2)
```


Parameter	Description
action_name	The name of the current action within the current module. When the current action terminates, call this procedure with the name of the next action if there is one, or null if there is not. Names longer than 32 bytes are truncated.

Table 3 – 3 Parameters for SET_ACTION Procedure

Setting the Client Information

To supply additional information about the client application, use the SET_CLIENT_INFO procedure in the DBMS_APPLICATION_INFO package.

Syntax

The parameter for the SET_CLIENT_INFO procedure is described in Table 3 – 4. The syntax for this procedure is shown below:

```
DBMS_APPLICATION_INFO.SET_CLIENT_INFO(client_info IN VARCHAR2)
```

Parameter	Description
client_info	Use this parameter to supply any additional information about the client application. This information is stored in the V\$SESSIONS view. Information exceeding 64 bytes is truncated.

Table 3 – 4 Parameters for SET_CLIENT_INFO Procedure

Retrieving Application Information

Module and action names for a registered application can be retrieved by querying V\$SQLAREA, or by calling the READ_MODULE procedure in the DBMS_APPLICATION_INFO package. Client information can be retrieved by querying the V\$SESSION view, or by calling the READ_CLIENT_INFO procedure in the DBMS_APPLICATION_INFO package.

Querying V\$SQLAREA

The following is a sample query illustrating the use of the MODULE and ACTION column of the V\$SQLAREA.

```
SELECT sql_text, disk_reads, module, action
FROM v$sqlarea
WHERE module = 'add_employee';
```

SQL_TEXT	DISK_READS	MODULE	ACTION
INSERT INTO emp (ename, empno, sal, mgr, job, hiredate, comm, deptno) VALUES (name, next.emp_seq, manager, title, SYSDATE, commission, department)	1	add_employee	insert into emp
1 row selected.			

READ_MODULE Syntax

The parameters for the READ_MODULE procedure are described in Table 3 – 5. The syntax for this procedure is shown below:

DBMS_APPLICATION_INFO.READ_MODULE(

module_name

OUT

VARCHAR2,

action_name

OUT

VARCHAR2)

Parameter	Description
module_name	The last value that the module name was set to by calling SET_MODULE.
action_name	The last value that the action name was set to by calling SET_ACTION or SET_MODULE

Table 3 – 5 Parameters for READ_MODULE Procedure

READ_CLIENT_INFO Syntax

The parameter for the READ_CLIENT_INFO procedure is described in Table 3 – 6. The syntax for this procedure is shown below:

DBMS_APPLICATION_INFO.READ_CLIENT_INFO(client_info OUT VARCHAR2)

Parameter	Description
client_info	The last client information value supplied to the SET_CLIENT_INFO procedure.

Table 3 – 6 Parameters for READ_CLIENT_INFO Procedure

SQL Processing

Before tuning the SQL in your applications, you should understand the Oracle Server SQL (Structured Query Language) processing scheme. This chapter introduces the basics of SQL processing and outlines the general phases through which each SQL statement goes. Topics include

- the stages of executing each type of SQL statement
 - DML processing
 - DDL processing
- guidelines for controlling transactions
- discrete transactions
- serializable transactions
- managing shared SQL and PL/SQL areas
- the optimizer
- parallel query processing

Although some Oracle tools and applications simplify or mask the use of SQL, all database operations are performed using SQL.

SQL Statement Execution

Figure 4 – 1 outlines the stages commonly used to process and execute a SQL statement. In some cases, Oracle might execute these steps in a slightly different order. For example, the DEFINE stage could occur just before the FETCH stage, depending on how you wrote your code.

The following sections describe each phase of SQL statement processing for each type of SQL statement. As you read this information, remember that for many Oracle tools, several of the phases are performed automatically. Most users need not be concerned with or aware of this level of detail. However, you might find this information useful when writing Oracle applications.

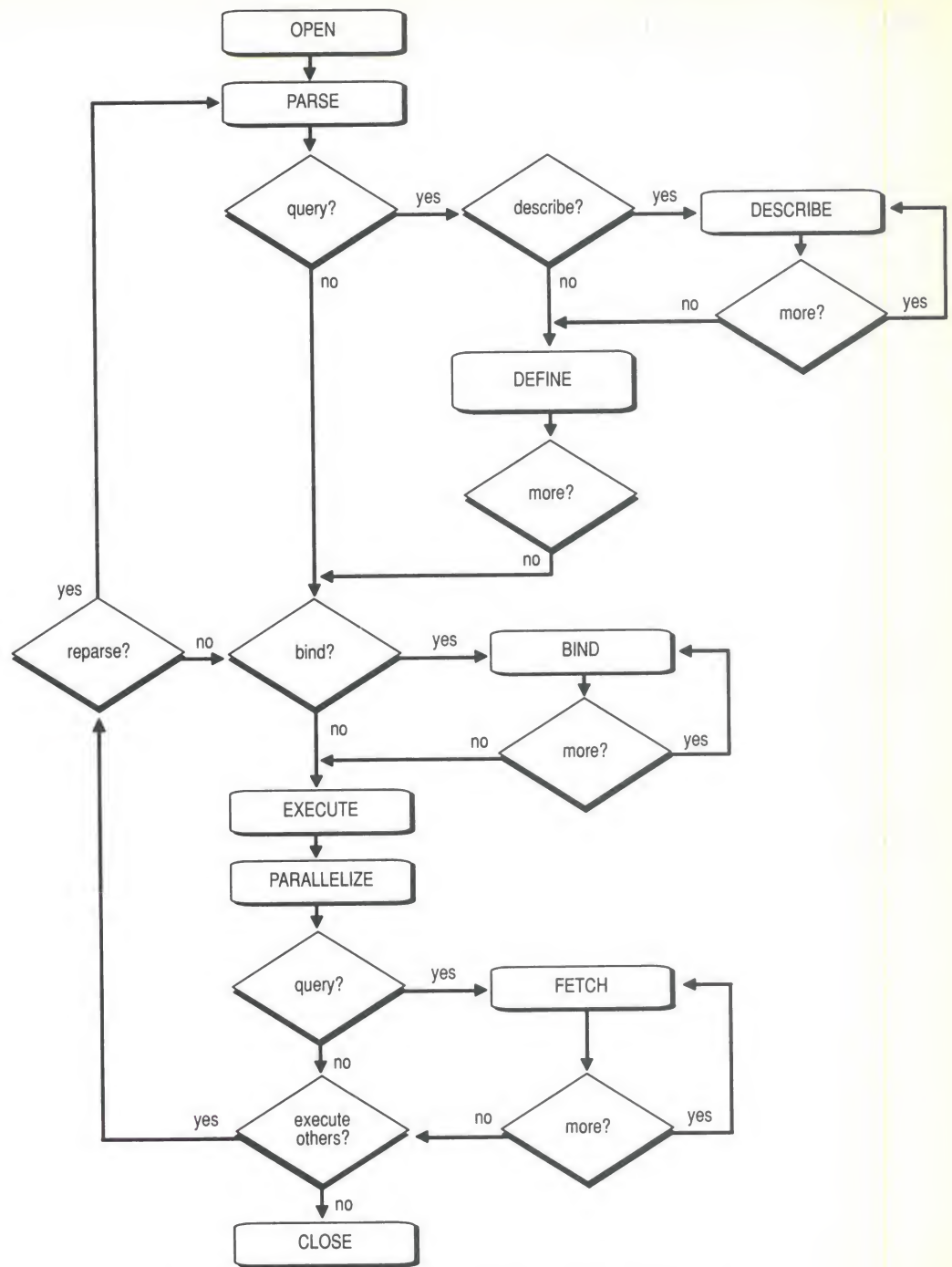


Figure 4 – 1 The Stages in Processing a SQL Statement

DML Statement Processing

This section describes a simplified look at what happens during the execution of a SQL statement. Queries (SELECTs) require additional steps as shown in Figure 4 – 1; refer to the section “Query Processing” on page 4 – 5 for more information.

Assume that you are using a Pro*C program to increase the salary for all employees in a department. Also assume that the program you are using has made a connection to Oracle and that you are connected to the proper schema to update the EMP table. You might embed the following SQL statement in your program:

```
EXEC SQL UPDATE emp SET sal = 1.10 * sal
      WHERE deptno = :dept_number;
```

DEPT_NUMBER is a program variable containing a value for department number. When the SQL statement is executed, the value of DEPT_NUMBER is used, as provided by the application program.

The next four sections explain what happens in each of the first four phases of DML statement processing. The same four phases are also necessary for each type of statement processing.

Stage 1 Create a Cursor

A program interface call creates a cursor. The cursor is created independently of any SQL statement; it is created in expectation of any SQL statement. In most applications, cursor creation is automatic. However, in precompiler programs, cursor creation can occur implicitly, or explicitly by declaring a cursor.

Stage 2 Parse the Statement

During parsing, the SQL statement is passed from the user process to Oracle and a parsed representation of the SQL statement is loaded into a shared SQL area. Many errors can be caught during this phase of statement processing. Parsing is the process of

- translating a SQL statement, verifying it to be a valid statement
- performing data dictionary lookups to check table and column definitions
- acquiring parse locks on required objects so that their definitions do not change during the statement’s parsing
- checking privileges to access referenced schema objects
- determining the optimal execution plan for the statement
- loading it into a shared SQL area
- for distributed statements, routing all or part of the statement to remote nodes that contain referenced data

A SQL statement is parsed only if a shared SQL area for an identical SQL statement does not exist in the shared pool. In this case, a new shared SQL area is allocated and the statement is parsed. For more information about shared SQL, refer to the section "Shared SQL" on page 4 – 13.

The parse phase includes processing requirements that need to be done only once no matter how many times the statement is executed. Oracle translates each SQL statement only once, re-executing that parsed statement during subsequent references to the statement.

Although the parsing of a SQL statement validates that statement, parsing only identifies errors that can be found **before statement execution**. Thus, certain errors cannot be caught by parsing. For example, errors in data conversion or errors in data (such as an attempt to enter duplicate values in a primary key) and deadlocks are all errors or situations that can only be encountered and reported during the execution phase.

Query Processing

Queries are different from other types of SQL statements because they return data as results if they are successful. Whereas other statements return simply success or failure, a query can return one row or thousands of rows. The results of a query are **always in tabular format**, and the rows of the result are *fetched* (retrieved), either a row at a time or in groups.

Several issues relate only to query processing. Queries include not only explicit SELECT statements but also the implicit queries in other SQL statements. For example, each of the following statements requires a query as a part of its execution:

```
INSERT INTO table SELECT ...  
UPDATE table SET x = y WHERE ...  
DELETE FROM table WHERE ...  
CREATE table AS SELECT ...
```

In particular, queries

- require *read consistency*
- can use temporary segments for intermediate processing
- can require the describe, define, and fetch phases of SQL statement processing

Stage 3
Describe Results

The describe phase is only necessary if the characteristics of a query's result are not known; for example, when a query is entered interactively by a user.

In this case, the describe phase is used to determine the characteristics (datatypes, lengths, and names) of a query's result.

Stage 4
Defining Output

In the define phase for queries, you specify the location, size, and datatype of variables defined to receive each fetched value. Oracle performs datatype conversion if necessary.

Stage 5
Bind Any Variables

At this point, Oracle knows the meaning of the SQL statement but still does not have enough information to execute the statement. Oracle needs values for any variables listed in the statement; in the example, Oracle needs a value for DEPT_NUMBER.

This process is called *binding variables*. A program must specify the location (memory address) where the value can be found. End users of applications might be unaware that they are specifying bind variables because the Oracle utility might simply prompt them for a new value.

Because you specify the location (binding by reference), you need not rebind the variable before re-execution. You can change its value and Oracle looks up the value on each execution, using the memory address.

Unless they are implied or defaulted, you must also specify a datatype and length for each value if Oracle needs to perform datatype conversion. For more information about specifying a datatype and length for a value, refer to the following publications:

- the *Programmer's Guide to the Oracle Precompilers* when using an Oracle precompiler (see "Dynamic SQL Method 4")
- the *Programmer's Guide to the Oracle Call Interface*

Stage 6
Execute the Statement

At this point, Oracle has all necessary information and resources, so the statement is executed. If the statement is a query or an INSERT statement, no rows need to be locked because no data is being changed. If the statement is an UPDATE or DELETE statement, however, all rows that the statement affects are locked from use by other users of the database, until the next COMMIT, ROLLBACK or SAVEPOINT for the transaction. This ensures data integrity.

For some statements you can specify a number of executions to be performed. This is called *array processing*. Given n number of executions, the bind and define locations are assumed to be the beginning of an array of size n .

Stage 7
Parallelize the Statement

When using the parallel query option, Oracle can parallelize queries and certain DDL operations. Parallelization causes multiple query servers to perform the work of the query so that the query can complete faster. Index creation and creating a table with a subquery can also be parallelized. Refer to Chapter 6, "Parallel Query Option", for more information on parallel query processing.

Stage 8
Fetch Rows of a Query Result

In the fetch phase, rows are selected and ordered (if requested by the query), and each successive fetch retrieves another row of the result, until the last row has been fetched.

DDL Statement Processing

The execution of DDL statements differs from the execution of DML statements and queries because the success of a DDL statement requires write access to the data dictionary. For these statements, the parse phase actually includes the parsing, data dictionary lookup, and execution.

Transaction management, session management, and system management SQL statements are processed using the parse and execute phases. To re-execute them, simply perform another execute.

Controlling Transactions

In general, only application designers using the programming interfaces to Oracle are concerned with which types of actions should be grouped together as one transaction. Transactions must be defined properly so work is accomplished in logical units and data is kept consistent. A transaction should consist of all of the necessary parts for one logical unit of work, no more and no less. Data in all referenced tables should be in a consistent state before the transaction begins and after it ends. Transactions should consist of only the SQL statements that comprise one consistent change to the data.

A transfer of funds between two accounts (the transaction or logical unit of work), for example, should include the debit to one account (one SQL statement) and the credit to another account (one SQL statement). Both actions should either fail or succeed together as a unit of work; the credit should not be committed without the debit. Other non-related actions, such as a new deposit to one account, should not be included in the transfer of funds transaction.

In addition to determining which types of actions form a transaction, when you design an application you must also determine when it is useful to use the `BEGIN_DISCRETE_TRANSACTION` procedure to improve the performance of short, non-distributed transactions.

Discrete Transactions

You can improve the performance of short, non-distributed transactions by using the `BEGIN_DISCRETE_TRANSACTION` procedure. This procedure streamlines transaction processing so short transactions can execute more rapidly.

Deciding When to Use Discrete Transactions

This streamlined transaction processing is useful for transactions that

- modify only a few database blocks
- never change an individual database block more than once per transaction
- do not modify data likely to be requested by long-running queries
- do not need to see the new value of data after modifying the data
- do not modify tables containing any LONG values

In deciding to use discrete transactions, you should consider the following factors:

- Can the transaction be designed to work within the constraints placed on discrete transactions?
- Does using discrete transactions result in a significant performance improvement under normal usage conditions?

Discrete transactions can be used concurrently with standard transactions. Choosing whether to use discrete transactions should be a part of your normal tuning procedure. Although discrete transactions can only be used for a subset of all transactions, for sophisticated users with advanced application requirements, where speed is the most critical factor, the performance improvements can make working within the design constraints worthwhile.

How Discrete Transactions Work

During a discrete transaction, all changes made to any data are deferred until the transaction commits. Redo information is generated, but is stored in a separate location in memory.

When the transaction issues a commit request, the redo information is written to the redo log file (along with other group commits) and the changes to the database block are applied directly to the block. The block is written to the database file in the usual manner. Control is returned to the application once the commit completes. This eliminates the need to generate undo information since the block is not actually modified until the transaction is committed, and the redo information is stored in the redo log buffers.

As with other transactions, the uncommitted changes of a discrete transaction are not visible to concurrent transactions. For regular transactions, undo information is used to re-create old versions of data for queries that require a consistent view of the data. Because no undo information is generated for discrete transactions, a discrete transaction that starts and completes during a long query can cause the query to receive the "snapshot too old" error if the query requests data changed by the discrete transaction. For this reason, you might want to avoid performing queries that access a large subset of a table that is modified by frequent discrete transactions.

To use the `BEGIN_DISCRETE_TRANSACTION` procedure, the `DISCRETE_TRANSACTIONS_ENABLED` initialization parameter must be set to `TRUE`. If this parameter is set to `FALSE`, all calls to `BEGIN_DISCRETE_TRANSACTION` are ignored and transactions requesting this service are handled as standard transactions. See *Oracle7 Server Reference* for more information about setting initialization parameters.

Errors During Discrete Transactions

Any errors encountered during processing of a discrete transaction cause the predefined exception `DISCRETE_TRANSACTION_FAILED` to be raised. These errors include the failure of a discrete transaction to comply with the usage notes outlined below. (For example, calling `BEGIN_DISCRETE_TRANSACTION` after a transaction has begun, or attempting to modify the same database block more than once during a transaction, raises the exception.)

Usage Notes

The `BEGIN_DISCRETE_TRANSACTION` procedure must be called before the first statement in a transaction. The call to this procedure is effective only for the duration of the transaction (that is, once the transaction is committed or rolled back, the next transaction is processed as a standard transaction).

Transactions that use this procedure cannot participate in distributed transactions.

Although discrete transactions cannot see their own changes, you can obtain the old value and lock the row, using the FOR UPDATE clause of the SELECT statement, before updating the value.

Because discrete transactions cannot see their own changes, a discrete transaction cannot perform inserts or updates on both tables involved in a referential integrity constraint.

For example, assume the EMP table has a foreign key constraint on the DEPTNO column that refers to the DEPT table. A discrete transaction cannot attempt to add a department into the DEPT table and then add an employee belonging to that department because the department is not added to the table until the transaction commits and the integrity constraint requires that the department exist before an insert into the EMP table can occur. These two operations must be performed in separate discrete transactions.

Because discrete transactions can change each database block only once, certain combinations of data manipulation statements on the same table are better suited for discrete transactions than others. One INSERT statement and one UPDATE statement used together are the least likely to affect the same block. Multiple UPDATE statements are also unlikely to affect the same block, depending on the size of the affected tables. Multiple INSERT statements (or INSERT statements that use queries to specify values), however, are likely to affect the same database block. Multiple DML operations performed on separate tables do not affect the same database blocks, unless the tables are clustered.

Example

An example of a transaction type that uses the BEGIN_DISCRETE_TRANSACTION procedure is an application for checking out library books. The following procedure is called by the library application with the book number as the argument. This procedure checks to see if the book is reserved before allowing it to be checked out. If more copies of the book have been reserved than are available, the status RES is returned to the library application, which calls another procedure to reserve the book, if desired. Otherwise, the book is checked out and the inventory of books available is updated.


```

CREATE PROCEDURE checkout (bookno IN NUMBER (10)
                           status OUT VARCHAR(5))
AS
DECLARE
    tot_books    NUMBER(3);
    checked_out  NUMBER(3);
    res          NUMBER(3);
BEGIN
    dbms_transaction.begin_discrete_transaction;
    FOR i IN 1 .. 2 LOOP
        BEGIN
            SELECT total, num_out, num_res
            INTO tot_books, checked_out, res
            FROM books
            WHERE book_num = bookno
            FOR UPDATE;
            IF res >= (tot_books - checked_out)
            THEN
                status := 'RES';
            ELSE
                UPDATE books SET num_out = checked_out + 1
                WHERE book_num = bookno;
                status := 'AVAIL'
            ENDIF;
            COMMIT;
            EXIT;
        EXCEPTION
            WHEN dbms_transaction.discrete_transaction_failed THEN
                ROLLBACK;
            END;
    END LOOP;
END;

```

Note the above loop construct. If the exception DISCRETE_TRANSACTION_FAILED occurs during the transaction, the transaction is rolled back, and the loop executes the transaction again. The second iteration of the loop is not a discrete transaction because the ROLLBACK statement ended the transaction; the next transaction processes as a standard transaction. This loop construct ensures that the same transaction is attempted again in the event of a discrete transaction failure.

Serializable Transactions

- Oracle allows application developers to set the isolation level of transactions. The isolation level determines what changes the transaction and other transactions can see. The ISO/ANSI SQL3 specification details the following levels of transaction isolation:
- SERIALIZABLE** transactions lose no updates, provide repeatable reads, and do not experience phantoms. Changes made to a serializable transaction are visible only to the transaction itself.
 - READ COMMITTED** transactions do not have repeatable reads and changes made in this transaction or other transactions are visible to all transactions. This is the default transaction isolation.

If you wish to set the transaction isolation level, you must do so before the transaction begins. Use the SET TRANSACTION ISOLATION level statement for a particular transaction, or the ALTER SESSION SET ISOLATION LEVEL statement for all subsequent transactions in the session. See *Oracle7 Server SQL Reference* for more information on the syntax of SET TRANSACTION and ALTER SESSION.

Shared SQL and PL/SQL

Oracle compares SQL statements and PL/SQL blocks issued directly by users and applications as well as recursive SQL statements issued internally by a DDL statement. If two identical statements are issued, the SQL or PL/SQL area used to process the first instance of the statement is *shared*, or used for the processing of the subsequent executions of that same statement.

Shared SQL and PL/SQL areas are shared memory areas; any Oracle process can use a shared SQL area. The use of shared SQL areas reduces memory usage on the database server, thereby increasing system throughput.

Shared SQL and PL/SQL areas are aged out of the shared pool by way of a least recently used algorithm (similar to database buffers). To improve performance and prevent reparsing, you may want to prevent large SQL or PL/SQL areas from aging out of the shared pool. The method for doing this is described in "Keeping Shared SQL and PL/SQL in the Shared Pool" on page 4 – 14.

Comparing SQL Statements and PL/SQL Blocks

Oracle automatically notices when two or more applications send identical SQL statements or PL/SQL blocks to the database. Oracle identifies identical statements using the following steps:

1. The text string of an issued statement is hashed. If the hash value is the same as a hash value for an existing SQL statement in the shared pool, Oracle proceeds to Step 2.
2. The text string of the issued statement, including case, blanks, and comments, is compared to all existing SQL statements identified in Step 1. For example:

```
SELECT * FROM emp; is identical to SELECT * FROM emp;  
SELECT * FROM emp; is not identical to SELECT * FROM Emp;
```

3. The objects referenced in the issued statement are compared to the referenced objects of all existing statements identified in Step 2. For example, if two users have EMP tables, the statement

```
SELECT * FROM emp;
```

is not considered identical because the statement references different tables for each user.

4. The bind types of bind variables used in a SQL statement must match.

Notice that Oracle does not have to parse a statement to determine if it is identical to another statement currently in the shared pool.

Keeping Shared SQL and PL/SQL in the Shared Pool

The DBMS_SHARED_POOL package allows objects to be kept in shared memory, so that they will not be aged out with the normal LRU mechanism. The DBMSPOOL.SQL and PRVTPPOOL.SQL procedure scripts create the package specification and package body for DBMS_SHARED_POOL.

By using the DBMS_SHARED_POOL package and by loading these SQL and PL/SQL areas early (before memory fragmentation occurs), the objects can be kept in memory, instead of aging out with the normal LRU mechanism. This procedure ensures that memory is available and prevents sudden, seemingly inexplicable slowdowns in user response time that occur when SQL and PL/SQL areas are accessed after aging out.

When to Use DBMS_SHARED_POOL

The procedures provided with the DBMS_SHARED_POOL package may be useful when loading large PL/SQL objects, such as the STANDARD and DIUTIL packages. When large PL/SQL objects are loaded, users' response time is affected because of the large number of smaller objects that need to be aged out from the shared pool to make room (due to memory fragmentation). In some cases, there may be insufficient memory to load the large objects.

DBMS_SHARED_POOL is also useful for frequently executed triggers. Because triggers are compiled each time they are aged out of the shared pool, you may want to keep compiled triggers on frequently used tables in the shared pool.

Usage Instructions

To use the DBMS_SHARED_POOL package to pin a SQL or PL/SQL area, complete the following steps.

1. Decide which packages/cursors you would like pinned in memory.
2. Startup the database.
3. Make a reference to the objects that causes them to be loaded. To pin a package, you can reference a dummy procedure defined in the package, or you can reference a package variable. To pin the cursor allocated for a SQL statement, execute the statement. To pin a trigger, issue a statement that causes the trigger to fire.
4. Make the call to DBMS_SHARED_POOL.KEEP to pin it.

This procedure ensures that the object is already loaded; otherwise, pinning may not be very useful. It also ensures that your system has not run out of the shared memory before the object is loaded. Finally, by pinning the object early in the life of the instance, this procedure prevents the memory fragmentation that could result from pinning a large chunk of memory in the middle of the shared pool.

Syntax	<p>The procedures provided with the DBMS_SHARED_POOL package are described below.</p>
DBMS_SHARED_POOL.SIZES	<p>This procedure shows the objects in the shared pool that are larger than the specified size.</p> <pre>dbms_shared_pool.sizes(minsize IN NUMBER)</pre> <p>Input Parameter:</p> <p>minsize</p> <p>Display objects in shared pool larger than this size, where size is measured in kilobytes.</p> <p>Output Parameters:</p> <p>To display the results of this procedure, before calling this procedure issue the following command using Server Manager or SQL*Plus:</p> <pre>SET SERVEROUTPUT ON SIZE minsize</pre> <p>You can use the results of this command as arguments to the KEEP or UNKEEP procedures.</p>
Example	<p>To show the objects in the shared pool that are larger than 2000 you would issue the following Server Manager or SQL*Plus commands:</p> <pre>SQL> SET SERVEROUTPUT ON SIZE 2000 SQL> EXECUTE DBMS_SHARED_POOL.SIZES(2000);</pre>
DBMS_SHARED_POOL.KEEP	<p>This procedure lets you keep an object in the shared pool. This procedure may not be supported in future releases.</p> <pre>dbms_shared_pool.keep(object IN VARCHAR2, [type IN CHAR DEFAULT P])</pre> <p>Input Parameters:</p> <p>object</p> <p>Either the parameter name or the cursor address of the object that you want kept in the shared pool. This value is the value displayed when you call the SIZES procedure.</p> <p>type</p> <p>The type of the object that you want kept in the shared pool. The types recognized are listed below:</p> <ul style="list-style-type: none"> P the object is a procedure C the object is a cursor R the object is a trigger

DBMS_SHARED_POOL.
UNKEEP

This procedure allows an object that you have requested to be kept in the shared pool to now be aged out of the shared pool. This procedure may not be supported in the future.

```
dbms_shared_pool.unkeep(object IN VARCHAR2,  
                        [type IN CHAR DEFAULT P])
```

Input Parameters:

object

Either the parameter name or the cursor address of the object that you no longer want kept in the shared pool. This value is the value displayed when you call the SIZES procedure.

type

The type of the object that you want aged out of the shared pool. The types recognized are listed below:

- P the object is a procedure
- C the object is a cursor
- R the object is a trigger

The Optimizer

When a SQL SELECT, UPDATE, INSERT, or DELETE statement is processed, Oracle must access the data referenced by the statement. The *optimizer* portion of Oracle is used to determine an efficient path to referenced data so that minimal I/O and processing time is required for statement execution. The optimizer formulates *execution plans* and chooses the most efficient plan before executing a statement. The optimizer uses one of two techniques to formulate execution plans for a SQL statement: a rule-based approach or a cost-based approach. Each of these approaches is described in Chapter 5, "The Optimizer".

Parallel Query Option

The parallel query option allows you to parallelize queries and certain DDL operations for faster performance. Parallelization is the process of breaking down the statement into smaller portions and executing each portion on a separate server process in parallel. Queries, index creation, and table creation with a subquery can all be parallelized. Chapter 6, "Parallel Query Option", describes parallel query execution.

The Optimizer: Overview

This chapter discusses how the Oracle optimizer chooses how to execute SQL statements. Topics include the following:

- what the Oracle optimizer is
- how Oracle optimizes SQL statements
- how to use histograms

What is Optimization?

Optimization is the process of choosing the most efficient way to execute a SQL statement. This is an important step in the processing of any Data Manipulation Language statement (SELECT, INSERT, UPDATE or DELETE). There may be many different ways for Oracle to execute such a statement, varying, for example, which tables or indexes are accessed in which order. The procedure used to execute a statement can greatly affect how quickly the statement executes. A part of Oracle called the *optimizer* chooses the way that it believes to be the most efficient.

The optimizer considers a number of factors to make what is usually the best choice among its alternatives. However, an application designer usually knows more about a particular application's data than the optimizer could know. Despite the best efforts of the optimizer, in some situations a developer can choose a more effective way to execute a SQL statement than the optimizer can.

Note: The optimizer may not make the same decisions from one version of Oracle to the next. In future versions of Oracle, the optimizer may make different decisions based on better, more sophisticated information available to it.

Execution Plans

To execute a Data Manipulation Language statement, Oracle may have to perform many steps. Each of these steps either physically retrieves rows of data from the database or prepares them in some way for the user issuing the statement. The combination of the steps Oracle uses to execute a statement is called an *execution plan*.

Example This example shows an execution plan for this SQL statement:

```
SELECT ename, job, sal, dname
  FROM emp, dept
 WHERE emp.deptno = dept.deptno
    AND NOT EXISTS
      (SELECT *
        FROM salgrade
       WHERE emp.sal BETWEEN losal AND hisal);
```

This statement selects the name, job, salary, and department name for all employees whose salaries do not fall into any recommended salary range.

Figure 5 – 1 shows a graphical representation of the execution plan.

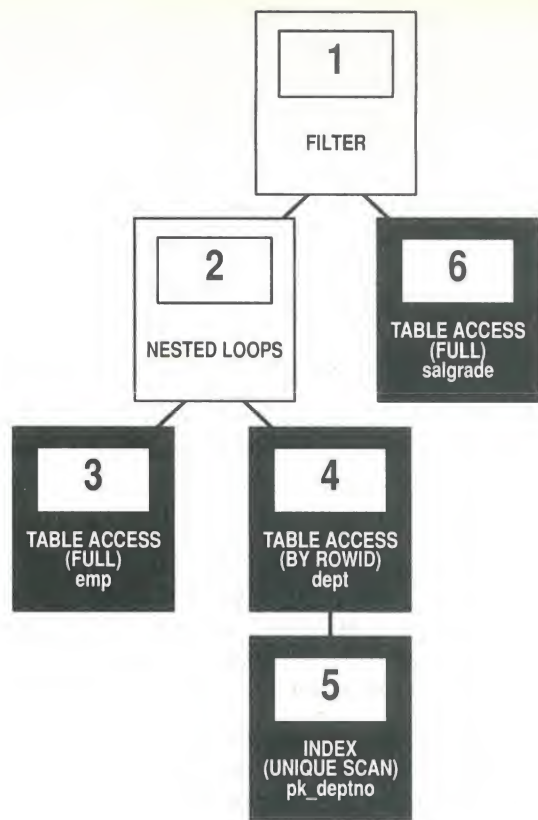


Figure 5 – 1 An Execution Plan

Steps of Execution Plan

Each step of the execution plan returns a set of rows that either are used by the next step or, in the last step, returned to the user or application issuing the SQL statement. A set of rows returned by a step is called a *row source*. Figure 5 – 1 is a hierarchical diagram showing the flow of rows from one step to another. The numbering of the steps reflects the order in which they are shown when you view the execution plan (how you view the execution will be explained shortly). This generally is **not** the order in which the steps are executed.

Each step of the execution plan either retrieves rows from the database or accepts rows from one or more row sources as input:

- Steps indicated by black boxes physically retrieve data from an object in the database. Such steps are called *access paths*:
 - Steps 3 and 6 read all the rows of the EMP and SALGRADE tables, respectively.
 - Step 5 looks up each DEPTNO value returned by step 3 in the PK_DEPTNO index. There it finds the ROWIDS of the associated rows in the DEPT table.
 - Step 4 retrieves from the DEPT table the rows whose ROWIDs were returned by Step 5.
- Steps indicated by white boxes operate on row sources:
 - Step 2 performs a nested loops operation, accepting row sources from Steps 3 and 4, joining each row from Step 3 source to its corresponding row in Step 4, and returning the resulting rows to Step 1.
 - Step 1 performs a filter operation. It accepts row sources from Steps 2 and 6, eliminates rows from Step 2 that have a corresponding row in Step 6, and returns the remaining rows from step 2 to the user or application issuing the statement.

Access paths are discussed in the section “Choosing Access Paths” on page 5 – 29. Methods by which Oracle joins row sources are discussed in the section “Join Operations” on page 5 – 47.

Order of Performing Execution Plan Steps

The steps of the execution plan are not performed in the order in which they are numbered. Oracle first performs the steps that appear as leaf nodes (for example, steps 3, 5, and 6) in the tree-structured graphical representation in Figure 5 – 1. The rows returned by each step become the row sources of its parent step. Then Oracle performs the parent steps.

To execute the statement for Figure 5 – 1, for example, Oracle performs the steps in this order:

- First, Oracle performs Step 3, and returns the resulting rows, one by one, to Step 2.
- For each row returned by Step 3, Oracle performs these steps:
 - Oracle performs Step 5 and returns the resulting ROWID to Step 4.
 - Oracle performs Step 4 and returns the resulting row to Step 2.

- Oracle performs Step 2, accepting a single row from Step 3 and a single row from Step 4, and returning a single row to Step 1.
- Oracle performs Step 6 and returns the resulting row, if any, to Step 1.
- Oracle performs Step 1. If a row is returned from Step 6, Oracle returns the row from Step 2 to the user issuing the SQL statement.

Note that Oracle performs Steps 5, 4, 2, 6, and 1 once for each row returned by Step 3. Many parent steps require only a single row from their child steps before they can be executed. For such a parent step, Oracle performs the parent step (and possibly the rest of the execution plan) as soon as a single row has been returned from the child step. If the parent of that parent step also can be activated by the return of a single row, then it is executed as well. Thus the execution can cascade up the tree, possibly to encompass the rest of the execution plan.

Oracle performs the parent step and all cascaded steps once for each row in turn retrieved by the child step. The parent steps that are triggered for each row returned by a child step include table accesses, index accesses, nested loops joins, and filters.

Some parent steps require all rows from their child steps before they can be performed. For such a parent step, Oracle cannot perform the parent step until all rows have been returned from the child step. Such parent steps include sorts, sort-merge joins, group functions, and aggregates.

The EXPLAIN PLAN Command

You can examine the execution plan chosen by the optimizer for a SQL statement by using the EXPLAIN PLAN command. This command causes the optimizer to choose the execution plan and then inserts data describing the plan into a database table. The following is such a description for the statement examined in the previous section:

ID	OPERATION	OPTIONS	OBJECT_NAME
0	SELECT STATEMENT		
1	FILTER		
2	NESTED LOOPS		
3	TABLE ACCESS	FULL	EMP
4	TABLE ACCESS	BY ROWID	DEPT
5	INDEX	UNIQUE SCAN	PK_DEPTNO
6	TABLE ACCESS	FULL	SALGRADE

You can obtain such a listing by using the EXPLAIN PLAN command and then querying the output table. For information on how to use this command and produce and interpret its output, see Appendix A, “Performance Diagnostic Tools”.

Each box in the figure and each row in the output table corresponds to a single step in the execution plan. For each row in the listing, the value in the ID column is the value shown in the corresponding box in the figure.

For a complete description of the EXPLAIN PLAN command, see Appendix A, “Performance Diagnostic Tools”.

Oracle’s Approaches to Optimization

To choose an execution plan for a SQL statement, the optimizer uses one of these approaches:

- | | |
|------------|---|
| rule-based | Using the <i>rule-based approach</i> , the optimizer chooses an execution plan based on the access paths available and the ranks of these access paths in Table 5 – 1 on page 5 – 31. |
| cost-based | Using the <i>cost-based approach</i> , the optimizer considers available access paths and factors in information based on statistics in the data dictionary for the objects (tables, clusters, or indexes) accessed by the statement to determine which execution plan is most efficient. The cost-based approach also considers <i>hints</i> , or optimization suggestions in the statement placed in a comment. |

The Rule-Based Approach

The rule-based approach chooses execution plans based on heuristically ranked operations. If there is more than one way to execute a SQL statement, the rule-based approach always uses the operation with the lower rank. Generally, operations of lower rank execute faster than those associated with constructs of higher rank.

The Cost-Based Approach

Conceptually, the cost-based approach consists of these steps:

1. The optimizer generates a set of potential execution plans for the statement based on its available access paths and hints.
2. The optimizer estimates the cost of each execution plan based on the data distribution and storage characteristics statistics for the tables, clusters, and indexes in the data dictionary.

The *cost* is an estimated value proportional to the expected resource usage needed to execute the statement using the execution plan. The optimizer calculates the cost based on the estimated computer resources, including but not limited to I/O, CPU time, and memory, required to execute the statement using the plan.

Non-parallel execution plans with greater costs take more time to execute than those with smaller costs. (When using the parallel query option, resource usage is not related to elapsed time.)

3. The optimizer compares the costs of the execution plans and chooses the one with the smallest cost.

Goal of the Cost-Based Approach By default, the goal of the cost-based approach is the best *throughput*, or minimal resource usage necessary to process all rows accessed by the statement.

Oracle can also optimize a statement with the goal of best *response time*, or minimal resource usage necessary to process the first row accessed by a SQL statement. For information on how the optimizer chooses an optimization approach and goal, see the section “Choosing an Optimization Approach and Goal” on page 5 – 27.

Note: Throughput and response times apply only to non-parallel statements. When using the parallel query option, resource usage is not related to elapsed time. See Chapter 6, “Parallel Query Option”, for more information about the parallel query option.

Statistics Used for the Cost-Based Approach The cost-based approach uses statistics to estimate the cost of each execution plan. These statistics quantify the data distribution and storage characteristics of tables, columns, and indexes. The ANALYZE command generates these statistics. Using these statistics, the optimizer estimates how much I/O, CPU time, and memory are required to execute a SQL statement using a particular execution plan.

The following data dictionary views display the statistics:

- USER_TABLES, ALL_TABLES, and DBA_TABLES
- USER_TAB_COLUMNS, ALL_TAB_COLUMNS, and DBA_TAB_COLUMNS
- USER_INDEXES, ALL_INDEXES, and DBA_INDEXES
- USER_CLUSTERS and DBA_CLUSTERS

For information on these statistics, see *Oracle7 Server Reference*.

Basics of the Cost-based Approach The plans generated by the cost-based optimizer depend on the sizes of the tables. When using the cost-based optimizer with a small amount of data to prototype an application, do not assume that the plan chosen for the full database will be the same as that chosen for the prototype.

The cost based optimizer assumes that a query will be executed on a multiuser system with a fairly low buffer cache hit rate. Thus a plan selected by the cost-based optimizer may not be the best plan for a single user system with a large buffer cache. Timing a query plan on a single user system with a large cache may not be a good predictor of performance for the same query on a busy multiuser system.

Analyzing a table uses more system resources than analyzing an index. It may be helpful to analyze the indexes for a table separately, with a higher sampling rate.

Use of access path and join method hints causes the cost-based optimizer to be invoked. Since the cost-based optimizer is dependent on statistics, it is important to analyze all tables referenced in a query which has hints, even though the rule-based optimizer may have been selected as the system default.

How Oracle Optimizes SQL Statements

This section explains how Oracle optimizes SQL statements. For any SQL statement processed by Oracle, the optimizer does the following:

evaluation of expressions and conditions	The optimizer first evaluates expressions and conditions containing constants as fully as possible.
statement transformation	For a complex statement involving, for example, correlated subqueries, the optimizer may transform the original statement into an equivalent join statement.
view merging	For a SQL statement that accesses a view, the optimizer often merges the query in the statement with that in the view and then optimizes the result.
choice of optimization approaches	The optimizer chooses either a rule-based or cost-based approach to optimization.
choice of access paths	For each table accessed by the statement, the optimizer chooses one or more of the available access paths to obtain the table's data.
choice of join orders	For a join statement that joins more than two tables, the optimizer chooses which pair of tables is joined first, and then which table is joined to the result, on so on.
choice of join operations	For any join statement, the optimizer chooses an operation to perform the join.

Types of SQL Statements

Oracle optimizes these different types of SQL statements:

simple statements	A <i>simple statement</i> is an INSERT, UPDATE, DELETE, or SELECT statement that involves only a single table.
simple queries	A <i>query</i> is another name for a SELECT statement.
joins	A <i>join</i> is a query that selects data from more than one table. A join is characterized by multiple tables in the FROM clause. Oracle pairs the rows from these tables using the condition specified in the WHERE clause and returns the resulting rows. This condition is called the <i>join condition</i> and usually compares columns of all the joined tables.

equijoins	An <i>equijoin</i> is characterized by a join condition containing an equality operator.
nonequijoins	A <i>nonequijoin</i> is characterized by a join condition containing something other than an equality operator.
outer joins	An <i>outer join</i> is characterized by a join condition that uses the outer join operator (+) with one or more of the columns of one of the tables. Oracle returns all rows that meet the join condition. Oracle also returns all rows from the table without the outer join operator for which there are no matching rows in the table with the outer join operator.
cartesian products	A join with no join condition results in a cartesian product, or a cross product. A cartesian product is the set of all possible combinations of rows drawn one from each table. In other words, for a join of two tables, each row in one table is matched in turn with every row in the other. A cartesian product for more than two tables is the result of pairing each row of one table with every row of the cartesian product of the remaining tables. All other kinds of joins are subsets of cartesian products effectively created by deriving the cartesian product and then excluding rows that fail the join condition.
complex statements	A complex statement is an INSERT, UPDATE, DELETE, or SELECT statement that contains a form of the SELECT statement called a <i>subquery</i> . This is a query within another statement that produces a set of values for further processing within the statement. The outer portion of the complex statement that contains a subquery is called the <i>parent statement</i> .
compound queries	A <i>compound query</i> is a query that uses set operators (UNION, UNION ALL, INTERSECT, or MINUS) to combine two or more simple or complex statements. Each simple or complex statement in a compound query is called a <i>component query</i> .
statements accessing views	You can also write simple, join, complex, and compound statements that access views as well as tables.

distributed
statements

A *distributed statement* is one that accesses data on a remote database.

**Evaluating Expressions
and Conditions**

The optimizer fully evaluates expressions whenever possible and translates certain syntactic constructs into equivalent constructs. This is done either because Oracle can more quickly evaluate the resulting expression than the original expression or because the original expression is merely a syntactic equivalent of the resulting expression. Since there are different SQL constructs that can operate identically (for example: = ANY (subquery) and IN (subquery)), Oracle maps these to a single construct.

Constants Any computation of constants is performed only once when the statement is optimized rather than each time the statement is executed. Consider these conditions that test for monthly salaries greater than 2000:

```
sal > 24000/12
sal > 2000
sal*12 > 24000
```

If a SQL statement contains the first condition, the optimizer simplifies it into the second condition.

Note that the optimizer does not simplify expressions across comparison operators. The optimizer does not simplify the third expression into the second. For this reason, application developers should write conditions that compare columns with constants whenever possible, rather than conditions with expressions involving columns.

LIKE The optimizer simplifies conditions that use the LIKE comparison operator to compare an expression with no wildcard characters into an equivalent condition that uses an equality operator instead. For example, the optimizer simplifies the first condition below into the second:

```
ename LIKE 'SMITH'
ename = 'SMITH'
```

The optimizer can simplify these expressions only when the comparison involves variable-length datatypes. For example, if ENAME was of type CHAR(10), the optimizer cannot transform the LIKE operation into an equality operation due to the comparison semantics of fixed-length datatypes.

IN The optimizer expands a condition that uses the IN comparison operator to an equivalent condition that uses equality comparison operators and OR logical operators. For example, the optimizer expands the first condition below into the second:

```
ename IN ('SMITH', 'KING', 'JONES')
ename = 'SMITH' OR ename = 'KING' OR ename = 'JONES'
```

ANY or SOME The optimizer expands a condition that uses the ANY or SOME comparison operator followed by a parenthesized list of values into an equivalent condition that uses equality comparison operators and OR logical operators. For example, the optimizer expands the first condition below into the second:

```
sal > ANY (:first_sal, :second_sal)
sal > :first_sal OR sal > :second_sal
```

The optimizer transforms a condition that uses the ANY or SOME operator followed by a subquery into a condition containing the EXISTS operator and a correlated subquery. For example, the optimizer transforms the first condition below into the second:

```
x > ANY (SELECT sal
        FROM emp
        WHERE job = 'ANALYST')
EXISTS (SELECT sal
        FROM emp
        WHERE job = 'ANALYST'
        AND x > sal)
```

ALL The optimizer expands a condition that uses the ALL comparison operator followed by a parenthesized list of values into an equivalent condition that uses equality comparison operators and AND logical operators. For example, the optimizer expands the first condition below into the second:

```
sal > ALL (:first_sal, :second_sal)
sal > :first_sal AND sal > :second_sal
```

The optimizer transforms a condition that uses the ALL comparison operator followed by a subquery into an equivalent condition that uses the ANY comparison operator and a complementary comparison operator. For example, the optimizer transforms the first condition below into the second:

```
x > ALL (SELECT sal
        FROM emp
        WHERE deptno = 10)
NOT (x <= ANY (SELECT sal
              FROM emp
              WHERE deptno = 10) )
```

The optimizer then transforms the second query into the following query using the rule for transforming conditions with the ANY comparison operator followed by a correlated subquery:

```
NOT EXISTS (SELECT sal
            FROM emp
            WHERE deptno = 10
            AND x <= sal)
```

BETWEEN The optimizer always replaces a condition that uses the BETWEEN comparison operator with an equivalent condition that uses the >= and <= comparison operators. For example, the optimizer replaces the first condition below with the second:

```
sal BETWEEN 2000 AND 3000
sal >= 2000 AND sal <= 3000
```

NOT The optimizer simplifies a condition to eliminate the NOT logical operator. The simplification involves removing the NOT logical operator and replacing a comparison operator with its opposite comparison operator. For example, the optimizer simplifies the first condition below into the second one:

```
NOT deptno = (SELECT deptno FROM emp WHERE ename = 'TAYLOR')
deptno <> (SELECT deptno FROM emp WHERE ename = 'TAYLOR')
```

Often a condition containing the NOT logical operator can be written many different ways. The optimizer attempts to transform such a condition so that the subconditions negated by NOTs are as simple as possible, even if the resulting condition contains more NOTs. For example, the optimizer simplifies the first condition below into the second and then into the third.

```
NOT (sal < 1000 OR comm IS NULL)
NOT sal < 1000 AND comm IS NOT NULL
sal >= 1000 AND comm IS NOT NULL
```

Transitivity If two conditions in the WHERE clause involve a common column, the optimizer can sometimes infer a third condition using the transitivity principle. The optimizer can then use the inferred condition to optimize the statement. The inferred condition could potentially make available an index access path that was not made available by the original conditions.

Imagine a WHERE clause containing two conditions of these forms:

```
WHERE column1 comp_oper constant
      AND column1 = column2
```

In this case, the optimizer infers the condition

```
column2 comp_oper constant
```


where:

<i>comp_oper</i>	Is any of the comparison operators =, !=, ^=, <, <>, < >, <=, or >=.
<i>constant</i>	Is any constant expression involving operators, SQL functions, literals, bind variables, and correlation variables.

Note: Transitivity is used only by the cost-based approach.

Example Consider this query in which the WHERE clause containing two conditions that each use the EMP.DEPTNO column:

```
SELECT *
  FROM emp, dept
 WHERE emp.deptno = 20
    AND emp.deptno = dept.deptno;
```

Using transitivity, the optimizer infers this condition:

```
dept.deptno = 20
```

If there is an index on the DEPT.DEPTNO column, this condition makes available access paths using that index.

Note: The optimizer only infers conditions that relate columns to constant expressions, rather than columns to other columns. Imagine a WHERE clause containing two conditions of these forms:

```
WHERE column1 comp_oper column3
    AND column1 = column2
```

In this case, the optimizer does not infer this condition:

```
column2 comp_oper column3
```

Transforming Statements

Since SQL is such a flexible query language, there often are many statements you could formulate to achieve a given goal. Sometimes the optimizer transforms one such statement into another that achieves the same goal if the second statement can be executed more efficiently. This section discusses the following transformations that the optimizer can make:

- transforming queries containing ORs into compound statements containing UNION ALL
- transforming complex statements into join statements

Transforming ORs into Compound Queries

If a query contains a WHERE clause with multiple conditions combined with OR operators, the optimizer transforms it into an equivalent compound query that uses the UNION ALL set operator if this will make it execute more efficiently:

- If each condition individually makes an index access path available, the optimizer can make the transformation. The optimizer then chooses an execution plan for the resulting statement that accesses the table multiple times using the different indexes and then puts the results together.
- If any condition requires a full table scan because it does not make an index available, the optimizer does not transform the statement. The optimizer chooses a full table scan to execute the statement, and Oracle tests each row in the table to determine whether it satisfies any of the conditions.

For information on access paths and how indexes make them available, see Table 5 – 1 on page 5 – 31 and the sections that follow it. For statements that use the cost-based approach, the optimizer may also use statistics to determine whether to make the transformation by estimating and then comparing the costs of executing the original statement versus the resulting statement.

Example

Consider this query with a WHERE clause that contains two conditions combined with an OR operator:

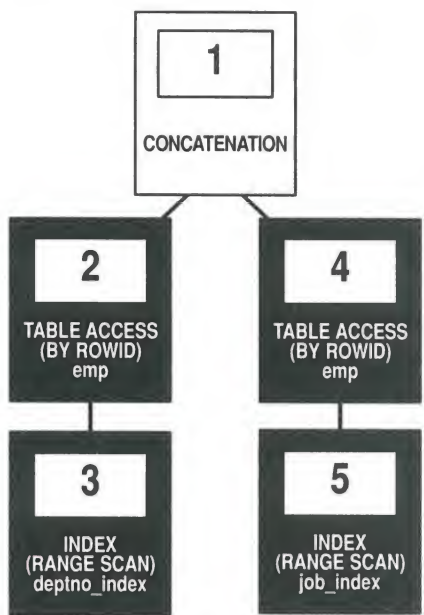
```
SELECT *  
  FROM emp  
 WHERE job = 'CLERK'  
    OR deptno = 10;
```

If there are indexes on both the JOB and DEPTNO columns, the optimizer may transform the query above into the equivalent query below:

```
SELECT *  
  FROM emp  
 WHERE job = 'CLERK'  
UNION ALL  
SELECT *  
  FROM emp  
 WHERE deptno = 10  
    AND job <> 'CLERK';
```

If you are using the rule-based approach, the optimizer makes this transformation because each component query of the resulting compound query can be executed using an index. The rule-based approach assumes that executing the compound query using two index scans is faster than executing the original query using a full table scan.

If you are using the cost-based approach, the optimizer compares the cost of executing the original query using a full table scan with that of executing the resulting query when deciding whether to make the transformation. The execution plan for the transformed statement might look like this:



To execute the transformed query, Oracle performs these steps:

- Steps 3 and 5 scan the indexes on the JOB and DEPTNO columns using the conditions of the component queries. These steps obtain ROWIDs of the rows that satisfy the component queries.
- Steps 2 and 4 use the ROWIDs from Steps 3 and 5 to locate the rows that satisfy each component query.
- Step 1 puts together the row sources returned by Steps 2 and 4.

If either of the JOB or DEPTNO columns is not indexed, the optimizer does not even consider the transformation because the resulting compound query would require a full table scan to execute one of its component queries. Executing the compound query with a full table scan in addition to an index scan could not possibly be faster than executing the original query with a full table scan.

Example Consider this query and assume there is an index on the ENAME column:

```
SELECT *
  FROM emp
 WHERE ename = 'SMITH'
        OR sal > comm;
```

Transforming the query above would result in the compound query below:

```
SELECT *
  FROM emp
 WHERE ename = 'SMITH'
UNION ALL
SELECT *
  FROM emp
 WHERE sal > comm;
```

Since the condition in the WHERE clause of the second component query (SAL > COMM) does not make an index available, the compound query requires a full table scan. For this reason, the optimizer does not consider making the transformation and chooses a full table scan to execute the original statement.

Optimizing Complex Statements

To optimize a complex statement, the optimizer chooses one of these alternatives:

- to transform the complex statement into an equivalent join statement and then optimize the join statement
- to optimize the complex statement as is

Transforming Complex Statements into Join Statements

The optimizer transforms a complex statement into a join statement whenever such a transformation results in a join statement that is guaranteed to return exactly the same rows as the complex statement. This transformation allows Oracle to execute the statement by taking advantage of join optimization techniques described in the section “Optimizing Join Statements” on page 5 – 47.

Consider this complex statement that selects all rows from the ACCOUNTS table whose owners appear in the CUSTOMERS table:

```
SELECT *
  FROM accounts
 WHERE custno IN
        (SELECT custno FROM customers);
```


If the CUSTNO column of the CUSTOMERS table is a primary key or has a UNIQUE constraint, the optimizer can transform the complex query into this join statement that is guaranteed to return the same data:

```
SELECT accounts.*
FROM accounts, customers
WHERE accounts.custno = customers.custno;
```

The execution plan for this statement might look like the following figure:

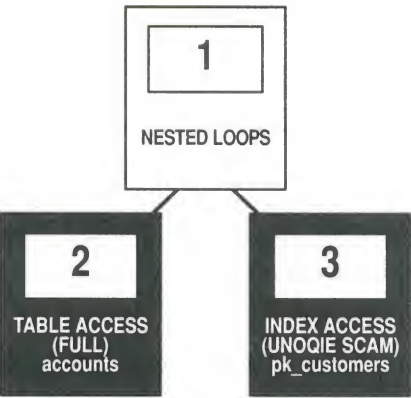


Figure 5 – 2 Execution Plan for a Nested Loops Join

To execute this statement, Oracle performs a nested loops join operation. For information on nested loops joins, see the section “Join Operations” on page 5 – 47.

If the optimizer cannot transform a complex statement into a join statement, the optimizer chooses execution plans for the parent statement and the subquery as though they were separate statements. Oracle then executes the subquery and uses the rows it returns to execute the parent query.

Consider this complex statement that returns all rows from the ACCOUNTS table that have balances that are greater than the average account balance:

```
SELECT *
FROM accounts
WHERE accounts.balance >
      (SELECT AVG(balance) FROM accounts);
```


There is no join statement that can perform the function of this statement, so the optimizer does not transform the statement. Note that complex queries whose subqueries contain group functions such as AVG cannot be transformed into join statements.

Optimizing Statements That Access Views

To optimize a statement that accesses a view, the optimizer chooses one of these alternatives:

- transforming the statement into an equivalent statement that accesses the view's base tables
- issuing the view's query, collecting all the returned rows, and then accessing this set of rows with the original statement as though it were a table

Transforming Statements That Access Views

To transform a statement that accesses a view into an equivalent statement that accesses the view's base tables, the optimizer can use one of these techniques:

- merging the view's query into the accessing statement
- merging the accessing statement into the view's query

The optimizer then optimizes the resulting statement.

Merging the View's Query into the Accessing Statement To merge the view's query into the accessing statement, the optimizer replaces the name of the view with the name of its base table in the accessing statement and adds the condition of the view's query's WHERE clause to the accessing statement's WHERE clause.

Example

Consider this view of all employees who work in department 10:

```
CREATE VIEW emp_10
AS SELECT empno,ename, job, mgr, hiredate, sal, comm, deptno
   FROM emp
   WHERE deptno = 10;
```

Consider this query that accesses the view. The query selects the IDs greater than 7800 of employees who work in department 10:

```
SELECT empno
   FROM emp_10
   WHERE empno > 7800;
```

The optimizer transforms the query into the following query that accesses the view's base table:

```
SELECT empno
   FROM emp
   WHERE deptno = 10
      AND empno > 7800;
```

If there are indexes on the DEPTNO or EMPNO columns, the resulting WHERE clause makes them available.

Merging the Accessing Statement into the View's Query The optimizer cannot always merge the view's query into the accessing statement. Such a transformation is not possible if the view's query contains

- set operators (UNION, UNION ALL, INTERSECT, MINUS)
- a GROUP BY clause
- a CONNECT BY clause
- a DISTINCT operator in the select list
- group functions (AVG, COUNT, MAX, MIN, SUM) in the select list

To optimize statements that access such views, the optimizer can merge the statement into the view's query.

Example Consider the TWO_EMP_TABLES view, which is the union of two employee tables. The view is defined with a compound query that uses the UNION set operator:

```
CREATE VIEW two_emp_tables
(empno, ename, job, mgr, hiredate, sal, comm, deptno) AS
SELECT empno, ename, job, mgr, hiredate, sal, comm,
deptno FROM emp1 UNION
SELECT empno, ename, job, mgr, hiredate, sal, comm,
deptno FROM emp2;
```

Consider this query that accesses the view. The query selects the IDs and names of all employees in either table who work in department 20:

```
SELECT empno, ename
FROM two_emp_tables
WHERE deptno = 20;
```

Since the view is defined as a compound query, the optimizer cannot merge the view query into the accessing query. Instead, the optimizer transforms the query by adding its WHERE clause condition into the compound query. The resulting statement looks like this:

```
SELECT empno, ename FROM emp1 WHERE deptno = 20
UNION
SELECT empno, ename FROM emp2 WHERE deptno = 20;
```

If there is an index on the DEPTNO column, the resulting WHERE clauses make it available.

Figure 5 – 3 shows the execution plan of the resulting statement.

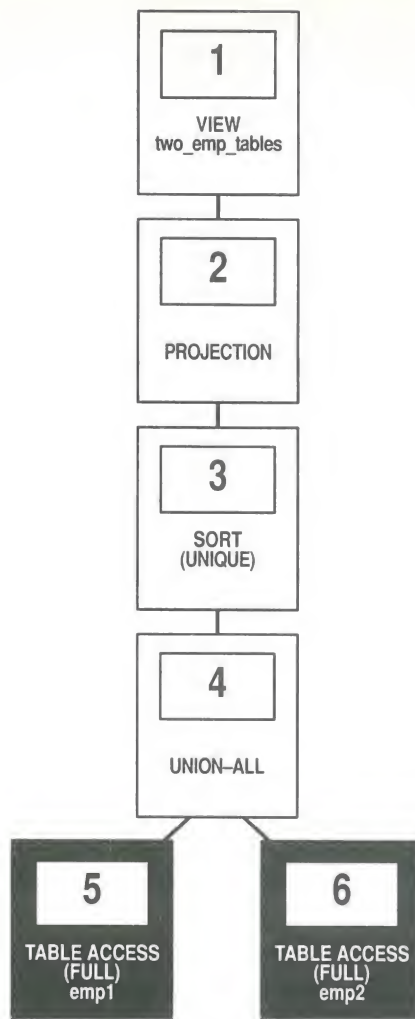


Figure 5 – 3 Accessing a View Defined with the UNION Set Operator

To execute this statement, Oracle performs these steps:

- Steps 5 and 6 perform full scans of the EMP1 and EMP2 tables.
- Step 4 performs a UNION-ALL operation returning all rows returned by either Step 5 or Step 6, including all copies of duplicates.
- Step 3 sorts the result of Step 4, eliminating duplicate rows.
- Step 2 extracts the desired columns from the result of Step 3.
- Step 1 indicates that the view's query was not merged into the accessing query.

Example Consider the view EMP_GROUP_BY_DEPTNO, which contains the department number, average salary, minimum salary, and maximum salary of all departments that have employees:

```
CREATE VIEW emp_group_by_deptno
AS SELECT deptno,
        AVG(sal) avg_sal,
        MIN(sal) min_sal,
        MAX(sal) max_sal
FROM emp
GROUP BY deptno;
```

Consider this query, which selects the average, minimum, and maximum salaries of department 10 from the EMP_GROUP_BY_DEPTNO view:

```
SELECT *
FROM emp_group_by_deptno
WHERE deptno = 10;
```

The optimizer transforms the statement by adding its WHERE clause condition into the view's query. The resulting statement looks like this:

```
SELECT deptno,
        AVG(sal) avg_sal,
        MIN(sal) min_sal,
        MAX(sal) max_sal,
FROM emp
WHERE deptno = 10
GROUP BY deptno;
```

If there is an index on the DEPTNO column, the resulting WHERE clause makes it available.

Figure 5 – 4 shows the execution plan for the resulting statement. The execution plan uses an index on the DEPTNO column.

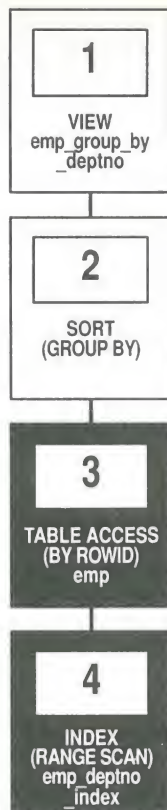


Figure 5 – 4 Accessing a View Defined with a GROUP BY Clause

To execute this statement, Oracle performs these operations:

- Step 4 performs a range scan on the index named EMP_DEPTNO_INDEX to retrieve the ROWIDs of all rows in the EMP table with a DEPTNO value of 10. EMP_DEPTNO_INDEX is an index on the DEPTNO column of the EMP table.
- Step 3 accesses the EMP table using the ROWIDs retrieved by Step 4.
- Step 2 sorts the rows returned by Step 3 to calculate the average, minimum, and maximum SAL values.
- Step 1 indicates that the view's query was not merged into the accessing query.

Example Consider this query, which accesses the EMP_GROUP_BY_DEPTNO view defined in the previous example. This query derives the averages for the average department salary, the minimum department salary, and the maximum department salary from the employee table:

```
SELECT AVG(avg_sal), AVG(min_sal), AVG(max_sal)
FROM emp_group_by_deptno;
```

The optimizer transforms this statement by applying the AVG group function to the select list of the view's query:

```
SELECT AVG(AVG(sal)), AVG(MIN(sal)), AVG(MAX(sal))
FROM emp
GROUP BY deptno;
```

Figure 5 – 5 shows the execution plan of the resulting statement:

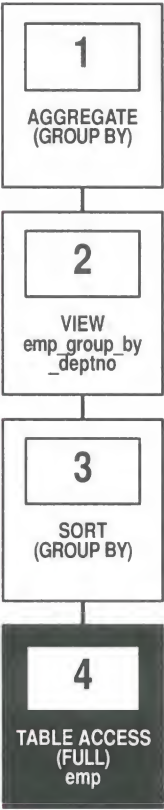


Figure 5 – 5 Applying Group Functions to a View Defined with a GROUP BY Clause

Optimizing Other Statements That Access Views

To execute this statement, Oracle performs these operations:

- Step 4 performs a full scan of the EMP table.
- Step 3 sorts the rows returned by Step 4 into groups based on their DEPTNO values and calculates the average, minimum, and maximum SAL value of each group.
- Step 2 indicates that the view's query was not merged into the accessing query.
- Step 1 calculates the averages of the values returned by Step 2.

The optimizer cannot transform all statements that access views into equivalent statements that access base table(s). To execute a statement that cannot be transformed, Oracle issues the view's query, collects the resulting set of rows, and then accesses this set of rows with the original statement as though it were a table.

Example

Consider the EMP_GROUP_BY_DEPTNO view defined in the previous section:

```
CREATE VIEW emp_group_by_deptno
AS SELECT deptno,
        AVG(sal) avg_sal,
        MIN(sal) min_sal,
        MAX(sal) max_sal
FROM emp
GROUP BY deptno;
```

Consider this query, which accesses the view. The query joins the average, minimum, and maximum salaries from each department represented in this view and to the name and location of the department in the DEPT table:

```
SELECT emp_group_by_deptno.deptno, avg_sal, min_sal,
        max_sal, dname, loc
FROM emp_group_by_deptno, dept
WHERE emp_group_by_deptno.deptno = dept.deptno;
```

Since there is no equivalent statement that accesses only base tables, the optimizer cannot transform this statement. Instead, the optimizer chooses an execution plan that issues the view's query and then uses the resulting set of rows as it would the rows resulting from a table access.

Figure 5 – 6 shows the execution plan for this statement.

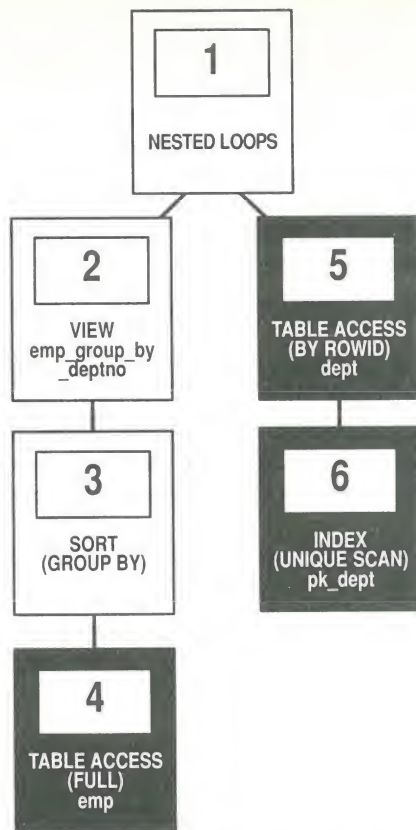


Figure 5 – 6 Joining a View Defined with a Group BY Clause to a Table

To execute this statement, Oracle performs these operations:

- Step 4 performs a full scan of the EMP table.
- Step 3 sorts the results of Step 4 and calculates the average, minimum, and maximum SAL values selected by the query for the EMP_GROUP_BY_DEPTNO view.
- Step 2 indicates that the previous two steps were executed to select data for a view.
- For each row returned by Step 2, Step 6 uses the DEPTNO value to perform a unique scan of the PK_DEPT index.
- Step 5 uses each ROWID returned by Step 6 to locate the row in the DEPTNO table with the matching DEPTNO value.
- Oracle combines each row returned by Step 2 with the matching row returned by Step 5 and returns the result.

Choosing an Optimization Approach and Goal

For more information on how Oracle performs a nested loops join operation, see the section “Join Operations” on page 5 – 47.

The optimizer’s behavior when choosing an optimization approach and goal for a SQL statement is affected by these factors:

- the OPTIMIZER_MODE initialization parameter
- statistics in the data dictionary
- the OPTIMIZER_GOAL parameter of the ALTER SESSION command
- hints in the statement

The OPTIMIZER_MODE Initialization Parameter

The OPTIMIZER_MODE initialization parameter establishes the default behavior for choosing an optimization approach for the instance. This parameter can have these values:

CHOOSE	This value causes the optimizer to choose between the rule-based approach and the cost-based approach based on whether the statistics used for the cost-based approach are present. If the data dictionary contains statistics for at least one of the accessed tables, the optimizer uses the cost-based approach and optimizes with a goal of best throughput. If data dictionary contains no statistics for any of the accessed tables, the optimizer uses the rule-based approach. This is the default value for the parameter.
RULE	This value causes the optimizer to choose the rule-based approach for all SQL statements issued to the instance regardless of the presence of statistics.
ALL_ROWS	This value causes the optimizer to use the cost-based approach for all SQL statements in the session regardless of the presence of statistics and to optimize with a goal of best throughput (minimum resource usage to complete the entire statement).
FIRST_ROWS	This value causes the optimizer to use the cost-based approach for all SQL statements in the session regardless of the presence of statistics and to optimize with a goal of best response time (minimum resource usage to return the first row of the result set).

Note that PL/SQL ignores the
OPTIMIZER_MODE=FIRST_ROWS initialization
parameter setting.

If the optimizer uses the cost-based approach for a SQL statement, and some tables accessed by the statement have no statistics, the optimizer uses internal information such as the number of data blocks allocated to these tables to estimate other statistics for these tables.

The OPTIMIZER_GOAL
Parameter of the ALTER
SESSION Command

The OPTIMIZER_GOAL parameter of the ALTER SESSION command can override the optimization approach and goal established by the OPTIMIZER_MODE initialization parameter for an individual session. This parameter can have these values:

- CHOOSE This value causes the optimizer to choose between the rule-based approach and the cost-based approach based on whether the statistics used for the cost-based approach are present. If the data dictionary contains statistics for at least one of the accessed tables, the optimizer uses the cost-based approach and optimizes with a goal of best throughput. If the data dictionary contains no statistics for any of the accessed tables, the optimizer uses the rule-based approach.
- ALL_ROWS This value causes the optimizer to use the cost-based approach for all SQL statements in the session regardless of the presence of statistics and to optimize with a goal of best throughput (minimum resource usage to complete the entire statement).
- FIRST_ROWS This value causes the optimizer to use the cost-based approach for all SQL statements in the session regardless of the presence of statistics and to optimize with a goal of best response time (minimum resource usage to return the first row of the result set).
- RULE This value causes the optimizer to choose the rule-based approach for all SQL statements issued to the instance regardless of the presence of statistics.

The value of this parameter affects the optimization of SQL statements issued by stored procedures and functions called during the session, but it does not affect the optimization of recursive SQL statements that Oracle issues during the session. The optimization approach for

recursive SQL statements is only affected by the value of the OPTIMIZER_MODE initialization parameter.

The FIRST_ROWS, ALL_ROWS, and RULE Hints

The FIRST_ROWS, ALL_ROWS, CHOOSE, and RULE hints can override the affects of the OPTIMIZER_MODE initialization parameter and the OPTIMIZER_GOAL parameter of the ALTER SESSION command for an individual SQL statement. For information on these hints, see Chapter 7, "Tuning SQL Statements".

Choosing Access Paths

One of the most important choices the optimizer makes when formulating an execution plan is how to retrieve the data from the database. For any row in any table accessed by a SQL statement, there may be many access paths by which that row can be located and retrieved. The optimizer chooses one of them.

This section discusses these topics:

- the basic methods by which Oracle can access data
- each access path and when it is available to the optimizer
- how the optimizer chooses among available access paths

Access Methods

This section describes basic methods by which Oracle can access data.

Full Table Scans A full table scan retrieves rows from a table. To perform a full table scan, Oracle reads all rows in the table, examining each row to determine whether it satisfies the statement's WHERE clause. Oracle reads every data block allocated to the table sequentially, so a full table scan can be performed very efficiently using multi-block reads. Oracle reads each data block only once.

Table Access by ROWID A table access by ROWID also retrieves rows from a table. The ROWID of a row specifies the datafile and data block containing the row and the location of the row in that block. Locating a row by its ROWID is the fastest way for Oracle to find a single row.

To access a table by ROWID, Oracle first obtains the ROWIDs of the selected rows, either from the statement's WHERE clause or through an index scan of one or more of the table's indexes. Oracle then locates each selected row in the table based on its ROWID.

Cluster Scans From a table stored in an indexed cluster, a cluster scan retrieves rows that have the same cluster key value. In an indexed cluster, all rows with the same cluster key value are stored in the same data blocks. To perform a cluster scan, Oracle first obtains the ROWID of one of the selected rows by scanning the cluster index. Oracle then locates the rows based on this ROWID.

Hash Scans Oracle can use a hash scan to locate rows in a hash cluster based on a hash value. In a hash cluster, all rows with the same hash value are stored in the same data blocks. To perform a hash scan, Oracle first obtains the hash value by applying a hash function to a cluster key value specified by the statement. Oracle then scans the data blocks containing rows with that hash value.

Index Scans An index scan retrieves data from an index based on the value of one or more columns of the index. To perform an index scan, Oracle searches the index for the indexed column values accessed by the statement. If the statement accesses only columns of the index, Oracle reads the indexed column values directly from the index, rather than from the table.

In addition to each indexed value, an index also contains the ROWIDs of rows in the table having that value. If the statement accesses other columns in addition to the indexed columns, Oracle then finds the rows in the table with a table access by ROWID or a cluster scan.

An index scan can be one of these types:

Unique	A unique scan of an index returns only a single ROWID. Oracle can only perform a unique scan in cases in which only a single ROWID, rather than many ROWIDs, is required. For example, Oracle performs a unique scan if there is a UNIQUE or a PRIMARY KEY constraint that guarantees that the statement accesses only a single row.
Range	A range scan of an index can return zero or more ROWIDs depending on how many rows are accessed by the statement.

Fast Full Index Scans The fast full index scan is an alternative to a full table scan when there is an index that contains all the columns that are needed for the query. FAST FULL SCAN is faster than a normal full index scan in that it can use multiblock I/O and can be parallelized just like a table scan. Unlike regular index scans, however, no keys can be used, and the rows will not necessarily come back in sorted order. If there is a predicate that could be used as start or stop key for the index, the optimizer will not consider a fast full scan.

The following query and plan illustrate this feature.

```
SELECT COUNT(*) FROM t1, t2
WHERE t1.c1 > 50 and t1.c2 = t2.c1;
```


The plan is as follows:

SELECT STATEMENT		
SORT		
HASH JOIN		
TABLE ACCESS	T1	FULL
INDEX	T2_C1_IDX	FAST FULL SCAN

Here, the fast full index scan can be used for table T2 since only column C1 is needed and there is no predicate that can be used as an index key. By contrast, FAST FULL SCAN could not be used for a nested loop join with the same join order. This is because the join predicate can be used as an index key for that type of join.

FAST FULL SCAN has a special index hint, INDEX_FFS, which has the same format and arguments as the regular INDEX hint.

You must set the V733_PLANS_ENABLED initialization parameter to TRUE for FAST FULL SCAN to be available.

Access Paths

Table 5 – 1 lists access paths. The rule-based approach uses the rank of each path to choose a path when more than one path is available.

Rank	Access Path
1	Single row by ROWID
2	Single row by cluster join
3	Single row by hash cluster key with unique or primary key
4	Single row by unique or primary key
5	Cluster join
6	Hash cluster key
7	Indexed cluster key
8	Composite key
9	Single-column indexes
10	Bounded range search on indexed columns
11	Unbounded range search on indexed columns
12	Sort-merge join
13	MAX or MIN of indexed column
14	ORDER BY on indexed columns
15	Full table scan

Table 5 – 1 Access Paths

The optimizer can only choose to use a particular access path for a table if the statement contains a WHERE clause condition or other construct that makes that access path available. Each of the following sections describes an access path and discusses

- when it is available
- the method Oracle uses to access data with it
- the output generated for it by the EXPLAIN PLAN command

Path 1 Single Row by ROWID This access path is only available if the statement's WHERE clause identifies the selected rows by ROWID or with the CURRENT OF CURSOR embedded SQL syntax supported by the Oracle Precompilers. To execute the statement, Oracle accesses the table by ROWID.

Example This access path is available in the following statement:

```
SELECT * FROM emp WHERE ROWID = '00000DC5.0000.0001';
```

The EXPLAIN PLAN output for this statement might look like this:

OPERATION	OPTIONS	OBJECT_NAME
SELECT STATEMENT		
TABLE ACCESS	BY ROWID	EMP

Path 2 Single Row by Cluster Join This access path is available for statements that join tables stored in the same cluster if both of these conditions are true:

- The statement's WHERE clause contains conditions that equate each column of the cluster key in one table with the corresponding column in the other table.
- The statement's WHERE clause also contains a condition that guarantees that the join returns only one row. Such a condition is likely to be an equality condition on the column(s) of a unique or primary key.

These conditions must be combined with AND operators. To execute the statement, Oracle performs a nested loops operation. For information on the nested loops operation, see the section "Join Operations" on page 5 – 47.

Example This access path is available for the following statement in which the EMP and DEPT tables are clustered on the DEPTNO column and the EMPNO column is the primary key of the EMP table:

```
SELECT *
FROM emp, dept
WHERE emp.deptno = dept.deptno
AND emp.empno = 7900;
```

The EXPLAIN PLAN output for this statement might look like this:

OPERATION	OPTIONS	OBJECT_NAME
SELECT STATEMENT		
NESTED LOOPS		
TABLE ACCESS	BY ROWID	EMP
INDEX	UNIQUE SCAN	PK_EMP
TABLE ACCESS	CLUSTER	DEPT

PK_EMP is the name of an index that enforces the primary key.

Path 3 **Single Row by Hash Cluster Key with Unique or Primary Key** This access path is available if both of these conditions are true:

- The statement’s WHERE clause uses all columns of a hash cluster key in equality conditions. For composite cluster keys, the equality conditions must be combined with AND operators.
- The statement is guaranteed to return only one row because the columns that make up the hash cluster key also make up a unique or primary key.

To execute the statement, Oracle applies the cluster’s hash function to the hash cluster key value specified in the statement to obtain a hash value. Oracle then uses the hash value to perform a hash scan on the table.

Example This access path is available in the following statement in which the ORDERS and LINE_ITEMS tables are stored in a hash cluster, and the ORDERNO column is both the cluster key and the primary key of the ORDERS table:

```
SELECT *
  FROM orders
 WHERE orderno = 65118968;
```

The EXPLAIN PLAN output for this statement might look like this:

OPERATION	OPTIONS	OBJECT_NAME
SELECT STATEMENT		
TABLE ACCESS	HASH	ORDERS

Path 4 Single Row by Unique or Primary Key This access path is available if the statement's WHERE clause uses all columns of a unique or primary key in equality conditions. For composite keys, the equality conditions must be combined with AND operators. To execute the statement, Oracle performs a unique scan on the index on the unique or primary key to retrieve a single ROWID and then accesses the table by that ROWID.

Example This access path is available in the following statement in which the EMPNO column is the primary key of the EMP table:

```
SELECT *
  FROM emp
 WHERE empno = 7900;
```

The EXPLAIN PLAN output for this statement might look like this:

OPERATION	OPTIONS	OBJECT_NAME
SELECT STATEMENT		
TABLE ACCESS	BY ROWID	EMP
INDEX	UNIQUE SCAN	PK_EMP

PK_EMP is the name of the index that enforces the primary key.

Path 5 Clustered Join This access path is available for statements that join tables stored in the same cluster if the statement's WHERE clause contains conditions that equate each column of the cluster key in one table with the corresponding column in the other table. For a composite cluster key, the equality conditions must be combined with AND operators. To execute the statement, Oracle performs a nested loops operation. For information on nested loops operations, see the section "Join Operations" on page 5 – 47.

Example This access path is available in the following statement in which the EMP and DEPT tables are clustered on the DEPTNO column:

```
SELECT *
  FROM emp, dept
 WHERE emp.deptno = dept.deptno;
```

The EXPLAIN PLAN output for this statement might look like this:

OPERATION	OPTIONS	OBJECT_NAME
SELECT STATEMENT		
NESTED LOOPS		
TABLE ACCESS	FULL	DEPT
TABLE ACCESS	CLUSTER EMP	

Path 6 Hash Cluster Key This access path is available if the statement's WHERE clause uses all the columns of a hash cluster key in equality conditions. For a composite cluster key, the equality conditions must be combined with AND operators. To execute the statement, Oracle applies the cluster's hash function to the hash cluster key value specified in the statement to obtain a hash value. Oracle then uses this hash value to perform a hash scan on the table.

Example This access path is available for the following statement in which the ORDERS and LINE_ITEMS tables are stored in a hash cluster and the ORDERNO column is the cluster key:

```
SELECT *
  FROM line_items
 WHERE deptno = 65118968;
```

The EXPLAIN PLAN output for this statement might look like this:

OPERATION	OPTIONS	OBJECT_NAME
<hr/>		
SELECT STATEMENT		
TABLE ACCESS	HASH	LINE_ITEMS

Path 7 Indexed Cluster Key This access path is available if the statement's WHERE clause uses all the columns of an indexed cluster key in equality conditions. For a composite cluster key, the equality conditions must be combined with AND operators. To execute the statement, Oracle performs a unique scan on the cluster index to retrieve the ROWID of one row with the specified cluster key value. Oracle then uses that ROWID to access the table with a cluster scan. Since all rows with the same cluster key value are stored together, the cluster scan requires only a single ROWID to find them all.

Example This access path is available in the following statement in which the EMP table is stored in an indexed cluster and the DEPTNO column is the cluster key:

```
SELECT * FROM emp
 WHERE deptno = 10;
```

The EXPLAIN PLAN output for this statement might look like this:

OPERATION	OPTIONS	OBJECT_NAME
<hr/>		
SELECT STATEMENT		
TABLE ACCESS	CLUSTER	EMP
INDEX	UNIQUE SCAN	PERS_INDEX

PERS_INDEX is the name of the cluster index.

Path 8 Composite Index This access path is available if the statement's WHERE clause uses all columns of a composite index in equality conditions combined with AND operators. To execute the statement, Oracle performs a range scan on the index to retrieve the ROWIDs of the selected rows and then accesses the table by those ROWIDs.

Example This access path is available in the following statement in which there is a composite index on the JOB and DEPTNO columns:

```
SELECT *
  FROM emp
 WHERE job = 'CLERK'
    AND deptno = 30;
```

The EXPLAIN PLAN output for this statement might look like this:

OPERATION	OPTIONS	OBJECT_NAME
SELECT STATEMENT		
TABLE ACCESS	BY ROWID	EMP
INDEX	RANGE SCAN	JOB_DEPTNO_INDEX

JOB_DEPTNO_INDEX is the name of the composite index on the JOB and DEPTNO columns.

Path 9 Single-Column Indexes This access path is available if the statement's WHERE clause uses the columns of one or more single-column indexes in equality conditions. For multiple single-column indexes, the conditions must be combined with AND operators.

If the WHERE clause uses the column of only one index, Oracle executes the statement by performing a range scan on the index to retrieve the ROWIDs of the selected rows and then accessing the table by these ROWIDs.

Example This access path is available in the following statement in which there is an index on the JOB column of the EMP table:

```
SELECT *
  FROM emp
 WHERE job = 'ANALYST';
```

The EXPLAIN PLAN output for this statement might look like this:

OPERATION	OPTIONS	OBJECT_NAME
SELECT STATEMENT		
TABLE ACCESS	BY ROWID	EMP
INDEX	RANGE SCAN	JOB_INDEX

JOB_INDEX is the index on EMP.JOB.

If the WHERE clause uses columns of many single-column indexes, Oracle executes the statement by performing a range scan on each index to retrieve the ROWIDs of the rows that satisfy each condition. Oracle then merges the sets of ROWIDs to obtain a set of ROWIDs of rows that satisfy all conditions. Oracle then accesses the table using these ROWIDs.

Oracle can merge up to five indexes. If the WHERE clause uses columns of more than five single-column indexes, Oracle merges five of them, accesses the table by ROWID, and then tests the resulting rows to determine whether they satisfy the remaining conditions before returning them.

Example This access path is available in the following statement in which there are indexes on both the JOB and DEPTNO columns of the EMP table:

```
SELECT *
  FROM emp
 WHERE job = 'ANALYST'
    AND deptno = 20;
```

The EXPLAIN PLAN output for this statement might look like this:

OPERATION	OPTIONS	OBJECT_NAME
SELECT STATEMENT		
TABLE ACCESS	BY ROWID	EMP
AND-EQUAL		
INDEX	RANGE SCAN	JOB_INDEX
INDEX	RANGE SCAN	DEPTNO_INDEX

The AND-EQUAL operation merges the ROWIDs obtained by the scans of the JOB_INDEX and the DEPTNO_INDEX, resulting in a set of ROWIDs of rows that satisfy the query.

Path 10 Bounded Range Search on Indexed Columns This access path is available if the statement's WHERE clause contains a condition that uses either the column of a single-column index or one or more columns that make up a leading portion of a composite index:

```
column = expr
column >[=] expr AND column <[=] expr
column BETWEEN expr AND expr
column LIKE 'c%'
```

Each of these conditions specifies a bounded range of indexed values that are accessed by the statement. The range is said to be bounded because the conditions specify both its least value and its greatest value. To execute such a statement, Oracle performs a range scan on the index and then accesses the table by ROWID.

This access path is not available if expr references the indexed column.

Example This access path is available in this statement in which there is an index on the SAL column of the EMP table:

```
SELECT *
  FROM emp
 WHERE sal BETWEEN 2000 AND 3000;
```

The EXPLAIN PLAN output for this statement might look like this:

OPERATION	OPTIONS	OBJECT_NAME
SELECT STATEMENT		
TABLE ACCESS	BY ROWID	EMP
INDEX	RANGE SCAN	SAL_INDEX

SAL_INDEX is the name of the index on EMP.SAL.

Example This access path is also available in the following statement in which there is an index on the ENAME column of the EMP table:

```
SELECT *
  FROM emp
 WHERE ename LIKE 'S%';
```

Path 11 Unbounded Range Search on Indexed Columns This access path is available if the statement’s WHERE clause contains one of these conditions that use either the column of a single-column index or one or more columns of a leading portion of a composite index:

```
WHERE column >[=] expr
WHERE column <[=] expr
```

Each of these conditions specify an unbounded range of index values accessed by the statement. The range is said to be unbounded because the condition specifies either its least value or its greatest value, but not both. To execute such a statement, Oracle performs a range scan on the index and then accesses the table by ROWID.

Example This access path is available in the following statement in which there is an index on the SAL column of the EMP table:

```
SELECT *
  FROM emp
 WHERE sal > 2000;
```


The EXPLAIN PLAN output for this statement might look like this:

OPERATION	OPTIONS	OBJECT_NAME
SELECT STATEMENT		
TABLE ACCESS	BY ROWID	EMP
INDEX	RANGE SCAN	SAL_INDEX

Example This access path is available in the following statement in which there is a composite index on the ORDER and LINE columns of the LINE_ITEMS table:

```
SELECT *
  FROM line_items
 WHERE order > 65118968;
```

The access path is available because the WHERE clause uses the ORDER column, a leading portion of the index.

Example This access path is not available in the following statement in which there is an index on the ORDER and LINE columns:

```
SELECT *
  FROM line_items
 WHERE line < 4;
```

The access path is not available because the WHERE clause only uses the LINE column, which is not a leading portion of the index.

Path 12 Sort–Merge Join This access path is available for statements that join tables that are not stored together in a cluster if the statement’s WHERE clause uses columns from each table in equality conditions. To execute such a statement, Oracle uses a sort–merge operation. Oracle can also use a nested loops operation to execute a join statement. For information on these operations and on when the optimizer chooses one over the other, see the section “Optimizing Join Statements” on page 5 – 47.

Example This access path is available for the following statement in which the EMP and DEPT tables are not stored in the same cluster:

```
SELECT *
  FROM emp, dept
 WHERE emp.deptno = dept.deptno;
```

The EXPLAIN PLAN output for this statement might look like this:

OPERATION	OPTIONS	OBJECT_NAME
SELECT STATEMENT		
MERGE JOIN		
SORT	JOIN	
TABLE ACCESS	FULL	EMP
SORT	JOIN	
TABLE ACCESS	FULL	DEPT

Path 13 MAX or MIN of Indexed Column This access path is available for a SELECT statement for which all of these conditions are true:

- The query uses the MAX or MIN function to select the maximum or minimum value of either the column of a single-column index or the leading column of a composite index. The index cannot be a cluster index.

The argument to the MAX or MIN function can be any expression involving the column, a constant, or the addition operator (+), the concatenation operation (||), or the CONCAT function.

- There are no other expressions in the select list.
- The statement has no WHERE clause or GROUP BY clause.

To execute the query, Oracle performs a range scan of the index to find the maximum or minimum indexed value. Since only this value is selected, Oracle need not access the table after scanning the index.

Example This access path is available for the following statement in which there is an index on the SAL column of the EMP table:

SELECT MAX(sal) FROM emp;

The EXPLAIN PLAN output for this statement might look like this:

OPERATION	OPTIONS	OBJECT_NAME
SELECT STATEMENT		
AGGREGATE	GROUP BY	
INDEX	RANGE SCAN	SAL_INDEX

Path 14 ORDER BY on Indexed Column This access path is available for a SELECT statement for which all of these conditions are true:

- The query contains an ORDER BY clause that uses either the column of a single-column index or a leading portion of a composite index. The index cannot be a cluster index.

- There must be a PRIMARY KEY or NOT NULL integrity constraint that guarantees that at least one of the indexed columns listed in the ORDER BY clause contains no nulls.
- The NLS_SORT parameter is set to BINARY.

To execute the query, Oracle performs a range scan of the index to retrieve the ROWIDs of the selected rows in sorted order. Oracle then accesses the table by these ROWIDs.

Example This access path is available for the following statement in which there is a primary key on the EMPNO column of the EMP table:

```
SELECT *
  FROM emp
 ORDER BY empno;
```

The EXPLAIN PLAN output for this statement might look like this:

OPERATION	OPTIONS	OBJECT_NAME
SELECT STATEMENT		
TABLE ACCESS	BY ROWID	EMP
INDEX	RANGE SCAN	PK_EMP

PK_EMP is the name of the index that enforces the primary key. The primary key ensures that the column does not contain nulls.

Path 15 Full Table Scan This access path is available for any SQL statement, regardless of its WHERE clause conditions.

This statement uses a full table scan to access the EMP table:

```
SELECT *
  FROM emp;
```

The EXPLAIN PLAN output for this statement might look like this:

OPERATION	OPTIONS	OBJECT_NAME
SELECT STATEMENT		
TABLE ACCESS	FULL	EMP

Note that these conditions do not make index access paths available:

- column1 > column2
- column1 < column2
- column1 >= column2
- column1 <= column2

where column1 and column2 are in the same table.

- column IS NULL
- column IS NOT NULL
- column NOT IN
- column != expr
- column LIKE '%pattern'

regardless of whether *column* is indexed.

- expr = expr2

where *expr* is an expression that operates on a column with an operator or function, regardless of whether the column is indexed.

- NOT EXISTS subquery
- any condition involving a column that is not indexed

Any SQL statement that contains only these constructs and no others that make index access paths available must use full table scans.

Choosing Among Access Paths

This section describes how the optimizer chooses among available access paths:

- when using the rule-based approach
- when using the cost-based approach

Choosing Among Access Paths with the Rule-Based Approach With the rule-based approach, the optimizer chooses whether to use an access path based on these factors:

- the available access paths for the statement
- the ranks of these access paths in Table 5 – 1 on page 5 – 31

To choose an access path, the optimizer first examines the conditions in the statement's WHERE clause to determine which access paths are available. The optimizer then chooses the most highly ranked available access path. Note that the full table scan is the lowest ranked access path on the list. This means that the rule-based approach always chooses an access path that uses an index if one is available, even if a full table scan might execute faster.

The order of the conditions in the WHERE clause does not normally affect the optimizer's choice among access paths.

Example

Consider this SQL statement, which selects the employee numbers of all employees in the EMP table with an ENAME value of 'CHUNG' and with a SAL value greater than 2000:


```

SELECT empno
FROM emp
WHERE ename = 'CHUNG'
AND sal > 2000;

```

Consider also that the EMP table has these integrity constraints and indexes:

- There is a PRIMARY KEY constraint on the EMPNO column that is enforced by the index PK_EMPNO.
- There is an index named ENAME_IND on the ENAME column.
- There is an index named SAL_IND on the SAL column.

Based on the conditions in the WHERE clause of the SQL statement, the integrity constraints, and the indexes, these access paths are available:

- A single-column index access path using the ENAME_IND index is made available by the condition ENAME = 'CHUNG'. This access path has rank 9.
- An unbounded range scan using the SAL_IND index is made available by the condition SAL > 2000. This access path has rank 11.
- A full table scan is automatically available for all SQL statements. This access path has rank 15.

Note that the PK_EMPNO index does not make the single row by primary key access path available because the indexed column does not appear in a condition in the WHERE clause.

Using the rule-based approach, the optimizer chooses the access path that uses the ENAME_IND index to execute this statement. The optimizer chooses this path because it is the most highly ranked path available.

Choosing Among Access Paths with the Cost-Based Approach With the cost-based approach, the optimizer chooses an access path based on these factors:

- the available access paths for the statement
- the estimated cost of executing the statement using each access path or combination of paths

To choose an access path, the optimizer first determines which access paths are available by examining the conditions in the statement's WHERE clause. The optimizer then generates a set of possible execution plans using available access paths and estimates the cost of

each plan using the statistics for the index, columns, and tables accessible to the statement. The optimizer then chooses the execution plan with the lowest estimated cost.

The optimizer's choice among available access paths can be overridden with hints. For information on hints, see Chapter 7, "Tuning SQL Statements".

To choose among available access paths, the optimizer considers these factors:

- **Selectivity:** The *selectivity* is the percentage of rows in the table that the query selects. Queries that select a small percentage of a table's rows have good selectivity, while a query that selects a large percentage of the rows has poor selectivity.

The optimizer is more likely to choose an index scan over a full table scan for a query with good selectivity than for one with poor selectivity. Index scans are usually more efficient than full table scans for queries that access only a small percentage of a table's rows, while full table scans are usually faster for queries that access a large percentage.

To determine the selectivity of a query, the optimizer considers these sources of information:

- the operators used in the WHERE clause
- unique and primary key columns used in the WHERE clause
- statistics for the table

The following examples in this section illustrate the way the optimizer uses selectivity.

- **DB_FILE_MULTIBLOCK_READ_COUNT:** Full table scans use multi-block reads, so the cost of a full table scan depends on the number of multi-block reads required to read the entire table, which depends on the number of blocks read by a single multi-block read, which is specified by the initialization parameter `DB_FILE_MULTIBLOCK_READ_COUNT`. For this reason, the optimizer may be more likely to choose a full table scan when the value of this parameter is high.

Example Consider this query, which uses an equality condition in its WHERE clause to select all employees named Jackson:

```
SELECT *  
  FROM emp  
 WHERE ename = 'JACKSON';
```

If the ENAME column is a unique or primary key, the optimizer determines that there is only one employee named Jackson, and the query returns only one row. In this case, the query is very selective, and the optimizer is most likely to access the table using a unique scan on the index that enforces the unique or primary key (access path 4).

Example Consider again the query in the previous example. If the ENAME column is not a unique or primary key, the optimizer can use these statistics to estimate the query's selectivity:

USER_TAB_COLUMNS. NUM_DISTINCT	This statistic is the number of values for each column in the table.
USER_TABLES.NUM_ROWS	This statistic is the number of rows in each table.

By dividing the number of rows in the EMP table by the number of distinct values in the ENAME column, the optimizer estimates what percentage of the employees have the same name. By assuming that the ENAME values are uniformly distributed, the optimizer uses this percentage as the estimated selectivity of the query.

Example Consider this query, which selects all employees with employee ID numbers less than 7500:

```
SELECT *
  FROM emp
 WHERE empno < 7500;
```

To estimate the selectivity of the query, the optimizer uses the boundary value of 7500 in the WHERE clause condition and the values of the HIGH_VALUE and LOW_VALUE statistics for the EMPNO column if available. These statistics can be found in the USER_TAB_COLUMNS view. The optimizer assumes that EMPNO values are evenly distributed in the range between the lowest value and highest value. The optimizer then determines what percentage of this range is less than the value 7500 and uses this value as the estimated selectivity of the query.

Example Consider this query, which uses a bind variable rather than a literal value for the boundary value in the WHERE clause condition:

```
SELECT *
  FROM emp
 WHERE empno < :e1;
```


The optimizer does not know the value of the bind variable E1. Indeed, the value of E1 may be different for each execution of the query. For this reason, the optimizer cannot use the means described in the previous example to determine selectivity of this query. In this case, the optimizer heuristically guesses a small value for the selectivity of the column (because it is indexed). The optimizer makes this assumption whenever a bind variable is used as a boundary value in a condition with one of the operators $<$, $>$, $<=$, or $>=$.

The optimizer's treatment of bind variables can cause it to choose different execution plans for SQL statements that differ only in the use of bind variables rather than constants. In one case in which this difference may be especially apparent, the optimizer may choose different execution plans for an embedded SQL statement with a bind variable in an Oracle Precompiler program and the same SQL statement with a constant in SQL*Plus.

Example Consider this query, which uses two bind variables as boundary values in the condition with the BETWEEN operator:

```
SELECT *
  FROM emp
 WHERE empno BETWEEN :low_e AND :high_e;
```

The optimizer decomposes the BETWEEN condition into these two conditions:

```
empno >= :low_e
empno <= :high_e
```

The optimizer heuristically estimates a small selectivity for indexed columns in order to favor the use of the index.

Example Consider this query, which uses the BETWEEN operator to select all employees with employee ID numbers between 7500 and 7800:

```
SELECT *
  FROM emp
 WHERE empno BETWEEN 7500 AND 7800;
```

To determine the selectivity of this query, the optimizer decomposes the WHERE clause condition into these two conditions:

```
empno >= 7500
empno <= 7800
```

The optimizer estimates the individual selectivity of each condition using the means described in a previous example. The optimizer then uses these selectivities ($S1$ and $S2$) and the absolute value function

(ABS) in this formula to estimate the selectivity (S) of the BETWEEN condition:

$$S = ABS(S1 + S2 - 1)$$

Optimizing Join Statements

To choose an execution plan for a join statement, the optimizer must make three related choices.

access paths	As for simple statements, the optimizer must choose an access path to retrieve data from each table in the join statement.
join operations	To join each pair of row sources, Oracle must perform one of these operations: nested loops sort-merge cluster
join order	To execute a statement that joins more than two tables, Oracle joins two of the tables, and then joins the resulting row source to the next table. This process is continued until all tables are joined into the result.

Join Operations

To join two row sources, Oracle must perform one of these operations:

- nested loops
- sort-merge
- cluster
- hash join

Nested Loops Join To perform a nested loops join, Oracle follows these steps:

1. The optimizer chooses one of the tables as the *outer table*, or the *driving table*. The other table is called the *inner table*.
2. For each row in the outer table, Oracle finds all rows in the inner table that satisfy the join condition.
3. Oracle combines the data in each pair of rows that satisfy the join condition and returns the resulting rows.

Figure 5 – 7 shows the execution plan for this statement using a nested loops join:

```
SELECT *  
  FROM emp, dept  
 WHERE emp.deptno = dept.deptno;
```

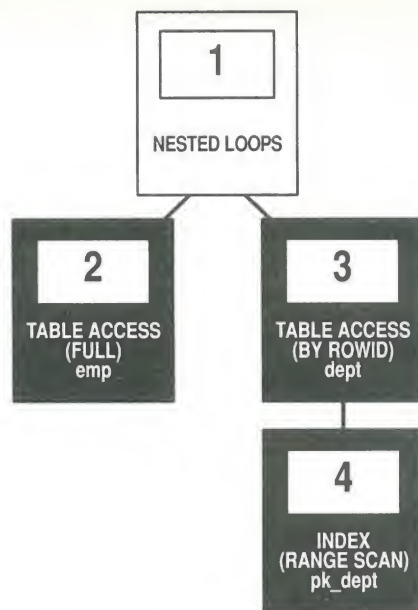


Figure 5 – 7 Nested Loops Join

To execute this statement, Oracle performs these steps:

- Step 2 accesses the outer table (EMP) with a full table scan.
- For each row returned by Step 2, Step 4 uses the EMP.DEPTNO value to perform a unique scan on the PK_DEPT index.
- Step 3 uses the ROWID from Step 4 to locate the matching row in the inner table (DEPT).
- Oracle combines each row returned by Step 2 with the matching row returned by Step 4 and returns the result.

Sort-Merge Join To perform a sort-merge join, Oracle follows these steps:

1. Oracle sorts each row source to be joined if they have not been sorted already by a previous operation. The rows are sorted on the values of the columns used in the join condition.
2. Oracle merges the two sources so that each pair of rows, one from each source, that contain matching values for the columns used in the join condition are combined and returned as the resulting row source.

Oracle can only perform a sort-merge join for an equijoin.

Figure 5 – 8 shows the execution plan for this statement using a sort-merge join:

```
SELECT *  
FROM emp, dept  
WHERE emp.deptno = dept.deptno;
```

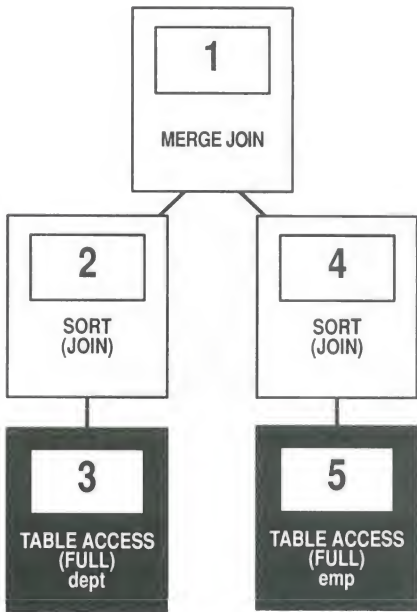


Figure 5 – 8 Sort-Merge Join

To execute this statement, Oracle performs these steps:

- Steps 3 and 5 perform full table scans of the EMP and DEPT tables.
- Steps 2 and 4 sort each row source separately.
- Step 1 merges the sources from Steps 2 and 4 together, combining each row from Step 2 with each matching row from Step 4 and returns the resulting row source.

Cluster Join Oracle can perform a cluster join only for an equijoin that equates the cluster key columns of two tables in the same cluster. In a cluster, rows from both tables with the same cluster key values are stored in the same blocks, so Oracle only accesses those blocks. For information on clusters, including how to decide which tables to cluster for best performance, see Chapter 7, “Tuning SQL Statements”.

Figure 5 – 9 shows the execution plan for this statement in which the EMP and DEPT tables are stored together in the same cluster:

```
SELECT *  
  FROM emp, dept  
 WHERE emp.deptno = dept.deptno;
```

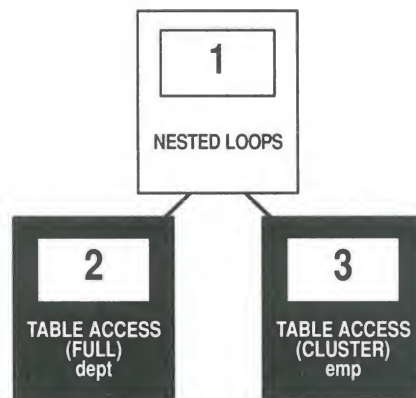


Figure 5 – 9 Cluster Join

To execute this statement, Oracle performs these steps:

- Step 2 accesses the outer table (DEPT) with a full table scan.
- For each row returned by Step 2, Step 3 uses the DEPT.DEPTNO value to find the matching rows in the inner table (EMP) with a cluster scan.

A cluster join is nothing more than a nested loops join involving two tables that are stored together in a cluster. Since each row from the DEPT table is stored in the same data blocks as the matching rows in the EMP table, Oracle can access matching rows most efficiently.

Hash Join To perform a hash join, Oracle follows these steps:

1. Oracle performs a full table scan on each of the tables and splits each into as many partitions as possible based on the available memory.
2. Oracle builds a hash table from one of the partitions (if possible, Oracle will select a partition that fits into available memory). Oracle then uses the corresponding partition in the other table to probe the hash table. All partitions pairs that do not fit into memory are placed onto disk.
3. For each pair of partitions (one from each table), Oracle uses the smaller one to build a hash table and the larger one to probe the hash table.

Oracle can only perform a hash join for an equijoin.

Figure 5 – 10 shows the execution plan for this statement using a hash join:

```
SELECT *
  FROM emp, dept
 WHERE emp.deptno = dept.deptno;
```

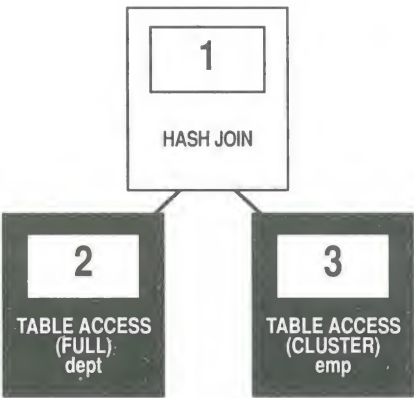


Figure 5 – 10 Hash Join

To execute this statement, Oracle performs these steps:

- Steps 2 and 3 perform full table scans of the EMP and DEPT tables.
- Step 1 builds a hash table out of the rows coming from 2 and probes it with each row coming from 3.

To enable the selection of the hash join algorithm you must use one of the following approaches: issue the statement `ALTER SESSION SET HASH_JOIN_ENABLED = TRUE`; or set the `HASH_JOIN_ENABLED` initialization parameter; or else employ the `USE_HASH` hint. See *Oracle7 Server SQL Reference* for more information on the syntax of the `ALTER SESSION` command.

The initialization parameter `HASH_AREA_SIZE` controls the memory to be used for hash join operations and the initialization parameter `HASH_MULTIBLOCK_IO_COUNT` controls the number of blocks a hash join operation should read and write concurrently. See *Oracle7 Server Reference* for more information about these initialization parameters.

This section describes how the optimizer chooses an execution plan for a join statement:

- when using the rule-based approach
- when using the cost-based approach

Note these considerations that apply to the cost-based and rule-based approaches:

- The optimizer first determines whether joining two or more of the tables definitely results in a row source containing at most one row. The optimizer recognizes such situations based on UNIQUE and PRIMARY KEY constraints on the tables. If such a situation exists, the optimizer places these tables first in the join order. The optimizer then optimizes the join of the remaining set of tables.
- For join statements with outer join conditions, the table with the outer join operator must come after the other table in the condition in the join order. The optimizer does not consider join orders that violate this rule.

Choosing Execution Plans for Joins with the Rule-Based Approach With the rule-based approach, the optimizer follows these steps to choose an execution plan for a statement that joins R tables:

1. The optimizer generates a set of R join orders, each with a different table as the first table. The optimizer generates each potential join order using this algorithm:
 - 1.1 To fill each position in the join order, the optimizer chooses the table with the most highly ranked available access path according to the ranks for access paths in Table 5 – 1 on page 5 – 31. The optimizer repeats this step to fill each subsequent position in the join order.
 - 1.2 For each table in the join order, the optimizer also chooses the operation with which to join the table to the previous table or row source in the order. The optimizer does this by “ranking” the sort-merge operation as access path 12 and applying these rules:
 - If the access path for the chosen table is ranked 11 or better, the optimizer chooses a nested loops operation using the previous table or row source in the join order as the outer table.
 - If the access path for the table is ranked lower than 12, and there is an equijoin condition between the chosen table and

the previous table or row source in join order, the optimizer chooses a sort–merge operation.

- If the access path for the chosen table is ranked lower than 12, and there is not an equijoin condition, the optimizer chooses a nested loops operation with the previous table or row source in the join order as the outer table.
2. The optimizer then chooses among the resulting set of execution plans. The goal of the optimizer’s choice is to maximize the number of nested loops join operations in which the inner table is accessed using an index scan. Since a nested loops join involves accessing the inner table many times, an index on the inner table can greatly improve the performance of a nested loops join. Usually, the optimizer does not consider the order in which tables appear in the FROM clause when choosing an execution plan.

The optimizer makes this choice by applying the following rules in order:

- 2.1 The optimizer chooses the execution plan with the fewest nested–loops operations in which the inner table is accessed with a full table scan.
- 2.2 If there is a tie, the optimizer chooses the execution plan with the fewest sort–merge operations.
- 2.3 If there is still a tie, the optimizer chooses the execution plan for which the first table in the join order has the most highly ranked access path:
 - If there is a tie among multiple plans whose first tables are accessed by the single–column indexes access path, the optimizer chooses the plan whose first table is accessed with the most merged indexes.
 - If there is a tie among multiple plans whose first tables are accessed by bounded range scans, the optimizer chooses the plan whose first table is accessed with the greatest number of leading columns of the composite index.
- 2.4 If there is still a tie, the optimizer chooses the execution plan for which the first table appears later in the query’s FROM clause.

Choosing Execution Plans for Joins with the Cost-Based Approach With the cost-based approach, the optimizer generates a set of execution plans based on the possible join orders, join operations, and available access paths. The optimizer then estimates the cost of each plan and chooses the one with the lowest cost. The optimizer estimates costs in these ways:

- The cost of a nested loops operation is based on the cost of reading each selected row of the outer table and each of its matching rows of the inner table into memory. The optimizer estimates these costs using the statistics in the data dictionary.
- The cost of a sort-merge join is based largely on the cost of reading all the sources into memory and sorting them.
- The optimizer also considers other factors when determining the cost of each operation. For example:
 - A smaller sort area size is likely to increase the cost for a sort-merge join because sorting takes more CPU time and I/O in a smaller sort area. Sort area size is specified by the initialization parameter `SORT_AREA_SIZE`.
 - A larger multi-block read count is likely to decrease the cost for a sort-merge join in relation to a nested loops join. If a large number of sequential blocks can be read from disk in a single I/O, an index on the inner table for the nested loops join is less likely to improve performance over a full table scan. The multi-block read count is specified by the initialization parameter `DB_FILE_MULTIBLOCK_READ_COUNT`.
 - For join statements with outer join conditions, the table with the outer join operator must come after the other table in the condition in the join order. The optimizer does not consider join orders that violate this rule.

With the cost-based approach, the optimizer's choice of join orders can be overridden with the `ORDERED` hint. If the `ORDERED` hint specifies a join order that violates the rule for outer join, the optimizer ignores the hint and chooses the order. You can also override the optimizer's choice of join operations with hints. For information on using hints, see Chapter 7, "Tuning SQL Statements".

Optimizing “Star”
Queries

Star queries are important in some data warehousing environments. Star schemas are normally characterized by a single large “fact” table and a number of much smaller reference tables. The fact table contains the primary information in the warehouse, such as all the pertinent information about sales over time. Each reference table contains information about the entries for a particular attribute in the fact table. For instance, there might be a reference table containing information about each product number which appears in the fact table. Thus there is a natural correspondence between the key column in the reference table (in this case, product number) and the product number column in the fact table.

In this example, the fact table might contain the columns Product_Num, Customer_ID, Date, Discount, Sales. The product reference table might contain the columns Product_Num, Product_Name, and Description. Reference tables might also exist for other attributes (such as Customer_ID).

It is common to join several of the reference tables, or subsets determined by predicates in the query, to the fact table. Because the fact table is so much larger than the reference tables, by far the most efficient execution of such a join is to construct a Cartesian product of the (filtered) reference tables and then perform a nested loops join of the result with the fact table. The fact table normally has a concatenated index to facilitate the join. Building the correct concatenated index on the “fact” table is critical for efficient execution of star queries.

The cost-based optimizer recognizes these star queries and generates efficient execution plans for them. Performance of star queries is benefited by the caching of intermediate results during the construction of the cartesian product. With release 7.3, star queries are no longer limited to 5 or fewer tables.

Star Query Example

This section discusses star query tuning, with reference to the following example:

```
SELECT SUM(dollars) FROM facts, time, product, market
WHERE market.stat = 'New York' AND
product.brand = 'MyBrand' AND
time.year = 1995 AND time.month = 'March' AND
/* Joins*/
time.key = facts.tkey AND
product.pkey = facts.pkey AND
market.mkey = facts.mkey;
```

A star query is thus a join between a very large table and a number of much smaller “lookup” tables. Each lookup table is joined to the large

<p>The Cost-Based Optimizer</p>	<p>table using a primary key to foreign key join, but the small tables are not joined to each other.</p> <p>Star queries are not recognized by the rule based optimizer. To execute star queries efficiently, you must use the cost based optimizer. Begin by using the ANALYZE command to gather statistics for each of the tables.</p>
<p>Indexing</p>	<p>The best way to execute a star query is usually to form the Cartesian product of the reduced rows from the small tables, and join this to the large table using a concatenated index. The index on the large table should include each of the join columns.</p> <p>In the example above, the index would be on the columns tkey, pkey, and mkey. The order of the columns in the index is critical to performance. the columns in the index should take advantage of any ordering of the data. If rows are added to the large table in time order, then tkey should be the first key in the index. When the data is a static extract from another database, it is worthwhile to sort the data on the key columns before loading it.</p> <p>If all queries specify predicates on each of the small tables, a single concatenated index suffices. If queries that omit leading columns of the concatenated index are frequent, additional indexes may be useful. In this example, if there are frequent queries that omit the time table, an index on pkey and mkey can be added.</p>
<p>Extended Star Schemas</p>	<p>Each of the small tables can be replaced by a join of several smaller tables. For example, the product table could be normalized into brand and manufacturer tables. Normalization of all of the small tables can cause performance problems. One problem is caused by the increased number of permutations that the optimizer must consider. The other problem is the result of multiple executions of the small table joins. Both problems can be solved by using denormalized views. For example,</p> <pre>CREATE VIEW prodview AS SELECT /*+ NO_MERGE */ * FROM brands, mfgs WHERE brands.mfkey = mfgs.mfkey;</pre> <p>This hint will both reduce the optimizer's search space, and cause caching of the result of the view.</p>

Hints

Usually, if you analyze the tables the optimizer will choose an efficient star plan. You can also use hints to improve the plan. The most precise method is to order the tables in the FROM clause in the order of the keys in the index, with the large table last. Then use the following hints:

```
/*+ ORDERED USE_NL(facts) INDEX(facts fact_concat) */
```

A more general method is to use the STAR hint `/*+ STAR */`, which will force the large table to be joined last using a nested loops join on the index. The optimizer will consider different permutations of the small tables.

Optimizing Compound Queries

To choose the execution plan for a compound query, the optimizer chooses an execution plan for each of its component queries and then combines the resulting row sources with the union, intersection, or minus operation, depending on the set operator used in the compound query.

Figure 5 – 11 shows the execution plan for this statement, which uses the UNION ALL operator to select all occurrences of all parts in either the ORDERS1 table or the ORDERS2 table:

```
SELECT part FROM orders1
UNION ALL
SELECT part FROM orders2;
```

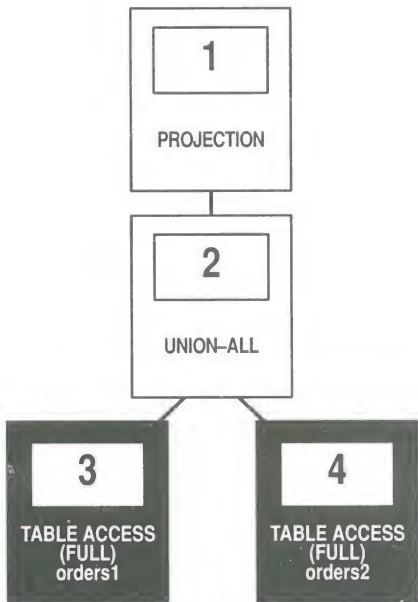


Figure 5 – 11 Compound Query with UNION ALL Set Operator

To execute this statement, Oracle performs these steps:

- Steps 3 and 4 perform full table scans on the ORDERS1 and ORDERS2 tables.
- Step 2 perform a UNION-ALL operation returning all rows that are returned by either Step 3 or Step 4 including all copies of duplicates.
- Step 1 performs an internal operation on the result of Step 2.

Figure 5 – 12 shows the execution plan for this statement, which uses the UNION operator to select all parts that appear in either the ORDERS1 table or the ORDERS2 table:

```
SELECT part FROM orders1
UNION
SELECT part FROM orders2;
```

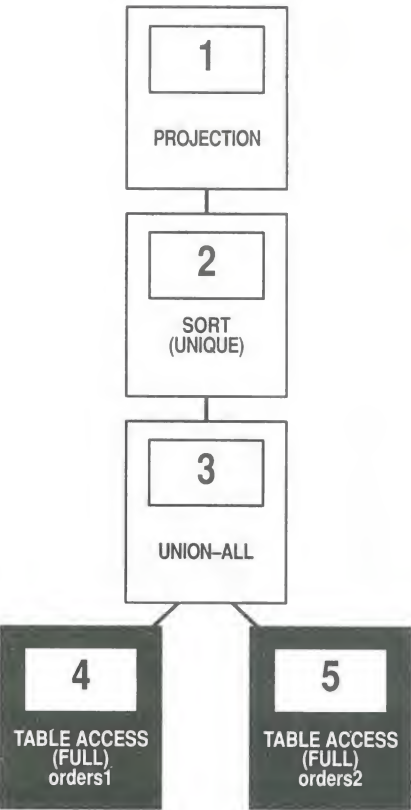


Figure 5 – 12 Compound Query with UNION Set Operator

This execution plan is identical to the one for the UNION-ALL operator shown in Figure 5 – 11, except that in this case Oracle uses the SORT operation to eliminate the duplicates returned by the UNION-ALL operation.

Figure 5 – 13 shows the execution plan for this statement, which uses the INTERSECT operator to select only those parts that appear in both the ORDERS1 and ORDERS2 tables:

```
SELECT part FROM orders1
INTERSECT
SELECT part FROM orders2;
```

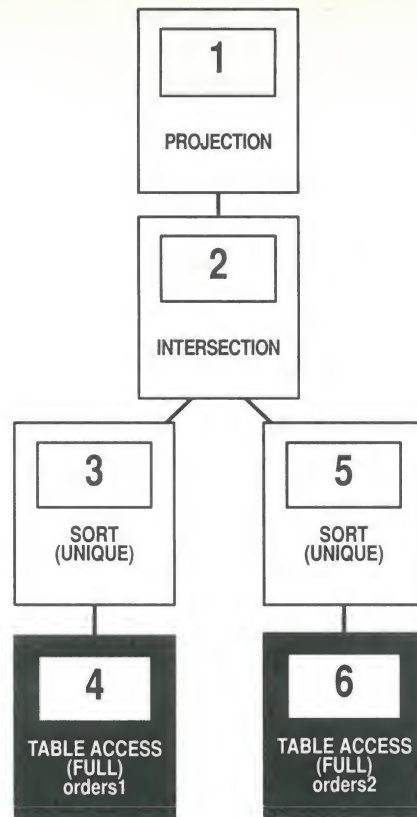


Figure 5 – 13 Compound Query with INTERSECT Set Operator

To execute this statement, Oracle performs these steps:

- Steps 4 and 6 perform full table scans of the ORDERS1 and ORDERS2 tables.
- Steps 3 and 5 sort the results of Steps 4 and 6, eliminating duplicates in each row source.
- Step 2 performs an INTERSECTION operation that returns only rows that are returned by both Steps 3 and 5.
- Step 1 performs an internal operation on the result of Step 2.

Optimizing Distributed Statements

The optimizer chooses execution plans for SQL statements that access data on remote databases in much the same way it chooses executions for statements that access only local data:

- If all the tables accessed by a SQL statement are collocated on the same remote database, Oracle sends the SQL statement to that remote database. The remote Oracle instance executes the statement and sends only the results back to the local database.
- If a SQL statement accesses tables that are located on different databases, Oracle decomposes the statement into individual fragments, each of which accesses tables on a single database. Oracle then sends each fragment to the database it accesses. The remote Oracle instance for each of these databases executes its fragment and returns the results to the local database, where the local Oracle instance may perform any additional processing the statement requires.

When choosing the execution plan for a distributed statement, the optimizer considers the available indexes on remote databases just as it does indexes on the local database. If the statement uses the cost-based approach, the optimizer also considers statistics on remote databases. Furthermore, the optimizer considers the location of data when estimating the cost of accessing it. For example, a full scan of a remote table has a greater estimated cost than a full scan of an identical local table.

Using Histograms

For uniformly distributed data, the cost-based approach makes fairly accurate guesses at the cost of executing a particular statement. However, when the data is not uniformly distributed, the optimizer cannot accurately estimate the selectivity of a query. For non-uniformly distributed data, Oracle allows you to store histograms describing the data distribution of a particular column.

Height-Balanced Histograms

Oracle's histograms are height balanced (as opposed to width balanced). Width-balanced histograms divide the data into a fixed number of equal-width ranges and then count the number of values falling into each range. Height-balanced histograms place the same number of values into each range so that the endpoints of the range are determined by how many values are in that range.

For example, suppose that the values in a single column of a 1000-row table range between 1 and 100, and suppose that you want a 10-bucket histogram (ranges in a histogram are often referred to as "buckets"). In a width-balanced histogram, the buckets would be of equal width (1-10, 11-20, 21-30, etc.) and each bucket would count the number of rows that fall into that range. In a height-balanced histogram, each bucket has the same height (in this case 100 rows), and then the endpoints for the buckets would be determined by the density of the distinct values in the column.

Advantages of Height-Balanced Histograms

The advantage of the height-balanced approach is clear when the data in the above example is highly skewed. Supposed that 800 rows of the example table had the value 5, and the remaining 200 rows are evenly distributed between 1 and 100. The width-balanced histogram would have 820 rows in the bucket labeled "1-10", and approximately 20 rows in the other buckets. The height-balanced histogram would have one bucket labeled "1-5", seven buckets labeled "5-5", one bucket labeled "5-50", and one bucket labeled "50-100".

If you wanted to know how many rows in the table contained the value "5", it is apparent from the height-balanced histogram that approximately 80% of the rows contain this value. However, the width-balanced histogram does not provide a mechanism for differentiating between the value "5" and the value "6". You would compute only 8% of the rows contain the value "5" in a width-balanced histogram. Thus, height-balanced histograms are more appropriate for determining the selectivity of column values.

When to Use Histograms

Histograms are stored in the dictionary and computed by using the `ANALYZE` command on a particular column. Therefore, there is a maintenance and space cost for using histograms. You should only compute histograms for columns which you know have highly-skewed data distribution.

Also, be aware that histograms, as well as all optimizer statistics, are static. If the data distribution of a column changes frequently, it is necessary to recompute the histogram for a given column.

Histograms are not useful for columns with the following characteristics:

- all predicates on the column use bind variables
- the column data is uniformly distributed
- the column is not used in `WHERE` clauses of queries
- the column is unique and is used only with equality predicates

How to Use Histograms

Create histograms on columns that are frequently used in `WHERE` clauses of queries and have a highly-skewed data distribution. You create a histogram by using the `ANALYZE TABLE` command. For example, if you want to create a 10-bucket histogram on the `SAL` column of the `EMP` table, issue the following statement:

```
ANALYZE TABLE emp COMPUTE STATISTICS FOR COLUMNS sal SIZE 10;
```

The `SIZE` keyword states the maximum number of buckets for the histogram. You would create a histogram on the `SAL` column if there were an unusual number of employees with the same salary and few employees with other salaries.

For more information about the `ANALYZE` command and its options, see the *Oracle7 Server SQL Reference* chapter of this manual.

Choosing the Number of Buckets for a Histogram

The default number of buckets is 75. This value provides an appropriate level of detail for most data distributions. However, since the number of buckets in the histogram, the sampling rate, and the data distribution all affect the usefulness of a histogram, you may need to experiment with different numbers of buckets to obtain the best results.

If the number of frequently occurring distinct values in a column is relatively small, then it is useful to set the number of buckets to be greater than the number of frequently occurring distinct values.

Viewing Histograms

You can find information about existing histograms in the database through the following data dictionary views: USER_HISTOGRAMS, ALL_HISTOGRAMS, and DBA_HISTOGRAMS. The number of buckets in each column's histogram is found in USER_TAB_COLUMNS, ALL_TAB_COLUMNS, and DBA_TAB_COLUMNS dictionary views. For more information and column descriptions of these views, see the *Oracle7 Server Reference* chapter of this manual.

Parallel Query Option

This chapter describes the Oracle Server parallel query option. This chapter discusses the following parallel query option topics:

- parallel query processing
- creating tables with subqueries in parallel
- parallelizing SQL statements
- setting the degree of parallelism
- managing query servers
- tuning for the parallel query option
- parallel index creation

The information in this chapter applies only to the Oracle Server with the parallel query option.

Note: This option is not the same as the Parallel Server option of the Oracle Server. The Parallel Server option is not required to use this feature. However, some aspects of the parallel query option apply only to an Oracle Parallel Server.

Parallel Query Processing

Without the parallel query option, the processing of a SQL statement is always performed by a single server process. With the parallel query option, multiple processes can work together simultaneously to process a single SQL statement. This capability is called *parallel query processing*. By dividing the work necessary to process a statement among multiple server processes, the Oracle Server can process the statement more quickly than if only a single server process processed it.

The parallel query option can dramatically improve performance for data-intensive operations associated with decision support applications or very large database environments. Symmetric multiprocessing (SMP), clustered, or massively parallel systems gain the largest performance benefits from the parallel query option because query processing can be effectively split up among many CPUs on a single system.

It is important to note that the query is parallelized dynamically at execution time. Thus, if the distribution or location of the data changes, Oracle automatically adapts to optimize the parallelization for each execution of a SQL statement.

The parallel query option helps systems scale in performance when adding hardware resources. If your system's CPUs and disk controllers are already heavily loaded, you need to alleviate the system's load before using the parallel query option to improve performance. The section "Tuning for the Parallel Query Option" on page 6 – 13 describes how your system can achieve the best performance with the parallel query option.

The Oracle Server can use parallel query processing for any of these statements:

- SELECT statements
- subqueries in UPDATE, INSERT, and DELETE statements
- CREATE TABLE ... AS SELECT statements
- CREATE INDEX

Parallel Query Process Architecture

Without the parallel query option, a server process performs all necessary processing for the execution of a SQL statement. For example, to perform a full table scan (for example, `SELECT * FROM EMP`), one process performs the entire operation. Figure 6 – 1 illustrates a server process performing a full table scan:

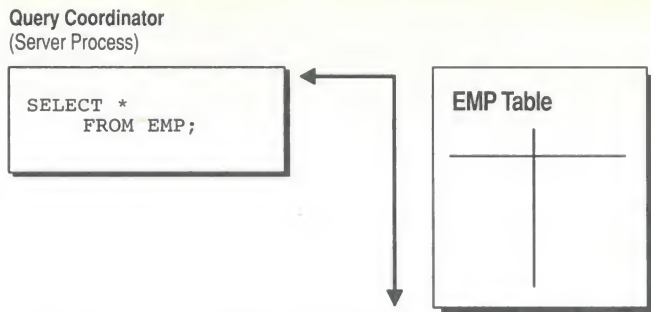


Figure 6 – 1 Full Table Scan without the Parallel Query Option

The parallel query option allows certain operations (for example, full table scans or sorts) to be performed in parallel by multiple query server processes. One process, known as the *query coordinator*, dispatches the execution of a statement to several *query servers* and coordinates the results from all of the servers to send the results back to the user. Figure 6 – 2 illustrates several query server processes simultaneously performing a partial scan of the EMP table. The results are then sent back to the query coordinator, which assembles the pieces into the desired full table scan.

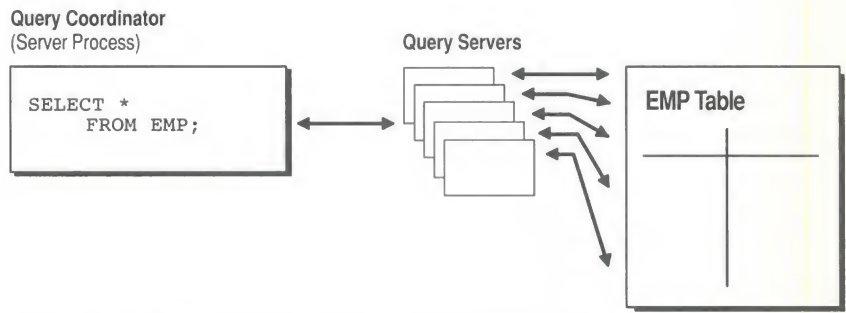


Figure 6 – 2 Multiple Query Servers Performing a Full Table Scan in Parallel

The query coordinator process is very similar to the server processes in previous releases of the Oracle Server. The difference is that the query coordinator can break down execution functions into parallel pieces and then integrate the partial results produced by the query servers. Query servers get assigned to each operation in a SQL statement (for example, a table scan or a sort operation), and the number of query servers assigned to a single operation is the *degree of parallelism* for a query.

The query coordinator calls upon the query servers during the execution of the SQL statement (not during the parsing of the statement). Therefore, when using the parallel query option with the multi-threaded server, the server processing the EXECUTE call of a user's statement becomes the query coordinator for the statement.

**CREATE TABLE ... AS
SELECT in Parallel**

Decision support applications often require large amounts of data to be summarized or "rolled up" into smaller tables for use with ad hoc, decision support queries. Rollup often must occur regularly (such as nightly or weekly) during a short period of system inactivity. Because the summary table is derived from other tables' data, the recoverability from media failure for the smaller table may or may not be important and can be turned off. The parallel query option allows you to parallelize the operation of creating a table as a subquery from another table or set of tables.

Figure 6 – 3 illustrates creating a table from a subquery in parallel.

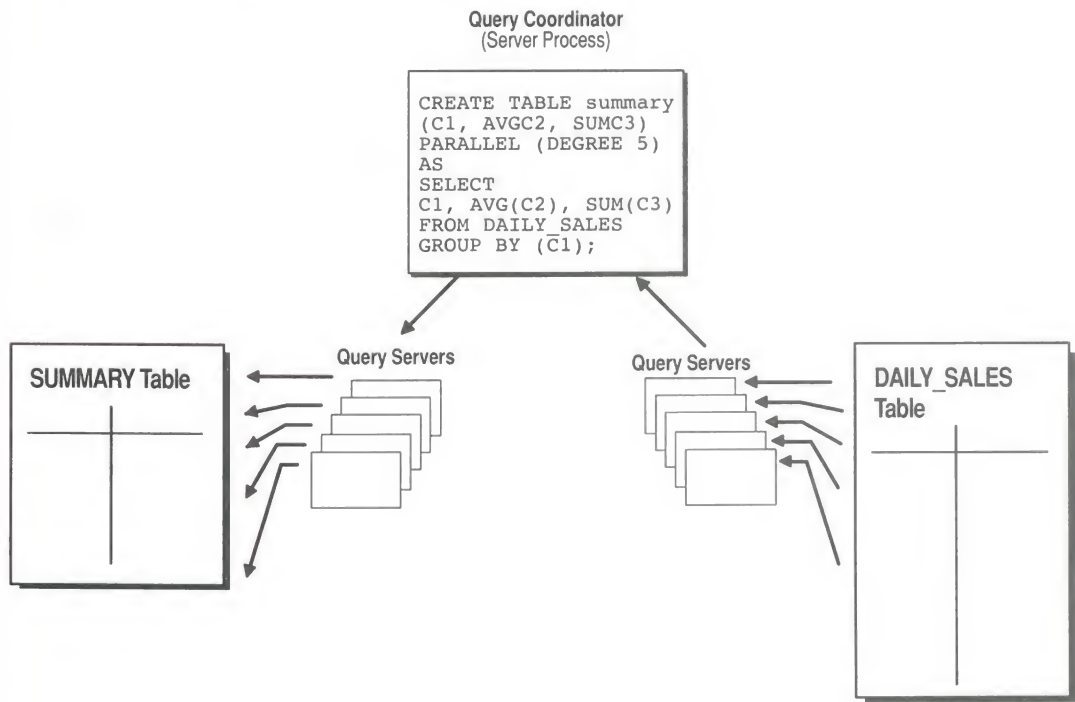


Figure 6 – 3 Creating a Summary Table in Parallel

If you disable recoverability during parallel table creation, you should take a backup of the tablespace containing the table once the table is created to avoid loss of the table due to media failure. For more information about recoverability of tables created in parallel, see the *Oracle7 Server Administrator's Guide*.

Clustered tables cannot be created and populated in parallel.

For a discussion of the syntax of the CREATE TABLE command, see the *Oracle7 Server SQL Reference*.

When creating a table in parallel, each of the query server processes uses the values in the STORAGE clause. Therefore, a table created with an INITIAL of 5M and a PARALLEL DEGREE of 12 consumes at least 60M of storage during table creation because each process starts with an extent of 5M. When the query coordinator process combines the extents, some of the extents may be trimmed, and the resulting table may be smaller than the requested 60M.

For more information on how extents are allocated when using the parallel query option, see *Oracle7 Server Concepts*.

Parallelizing SQL Statements

When a statement is parsed, the optimizer determines the execution plan of a statement. Optimization is discussed in Chapter 5, "The Optimizer". After the optimizer determines the execution plan of a statement, the query coordinator process determines the parallelization method of the statement. *Parallelization* is the process by which the query coordinator determines which operations can be performed in parallel and then enlists query server processes to execute the statement. This section tells you how the query coordinator process decides to parallelize a statement and how the user can specify how many query server processes can be assigned to each operation in an execution plan (that is, the *degree of parallelism*).

To decide how to parallelize a statement, the query coordinator process must decide whether to enlist query server processes and, if so, how many query server processes to enlist. When making these decisions, the query coordinator uses information specified in hints of a query, the table's definition, and initialization parameters. The precedence for selecting the degree of parallelism is described later in this section. It is important to note that the optimizer attempts to parallelize a query only if it contains at least one full table scan operation.

Parallelizing Operations

Each query undergoes an optimization and parallelization process when it is parsed. Therefore, when the data changes, if a more optimal execution plan or parallelization plan becomes available, Oracle can automatically adapt to the new situation.

In the case of creating a table in parallel, the subquery in the CREATE TABLE statement is parallelized and the actual population of the table is parallelized, as well as any enforcement of NOT NULL or CHECK constraints.

Before enlisting query server processes, the query coordinator process examines the operations in the execution plan to determine whether the individual operations can be parallelized. The Oracle Server can parallelize these operations:

- sorts
- joins
- table scans
- table population
- index creation

For a description of these and all operations, see Appendix A, “Performance Diagnostic Tools”.

Partitioning Rows to Each Query Server

The query coordinator process also examines the partitioning requirements of each operation. An operation’s *partitioning requirement* is the way in which the rows operated on by the operation must be divided, or partitioned, among the query server processes. The partitioning can be any of the following:

- range
- hash
- round robin
- random

After determining the partitioning requirement for each operation in the execution plan, the query coordinator determines the order in which the operations must be performed. With this information, the query coordinator determines the data flow of the statement. Figure 6 – 4 illustrates the data flow of the following query:

```
SELECT dname, MAX(sal), AVG(sal)
FROM emp, dept
WHERE emp.deptno = dept.deptno
GROUP BY dname;
```

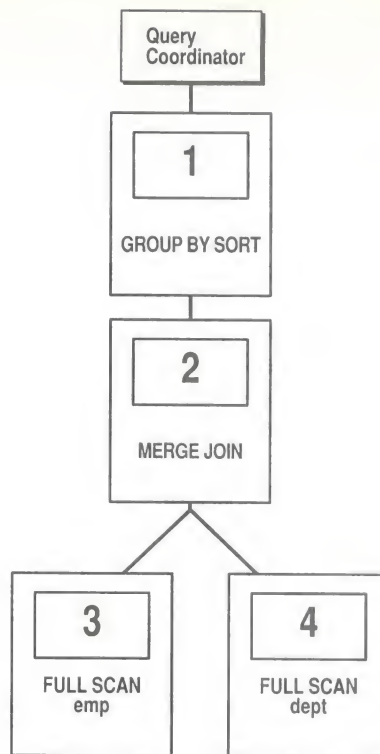



Figure 6 – 4 Data Flow Diagram for a Join of the EMP and DEPT Tables

Operations that require the output of other operations are known as *parent* operations. In Figure 6 – 4 the GROUP BY SORT operation is the parent of the MERGE JOIN operation because GROUP BY SORT requires the MERGE JOIN output.

Parallelism Between Operations

Parent operations can begin processing rows as soon as the child operations have produced rows for the parent operation to consume. In the previous example, while the query servers are producing rows in the FULL SCAN DEPT operation, another set of query servers can begin to perform the MERGE JOIN operation to consume the rows. When the FULL SCAN DEPT operation is complete, the FULL SCAN EMP operation can begin to produce rows.

Each of the two operations performed concurrently is given its own set of query server processes. Therefore, both query operations and the data flow tree itself have degrees of parallelism. The degree of parallelism of an individual operation is called *intra-operation* parallelism and the degree of parallelism between operations in a data flow tree is called *inter-operation* parallelism.

Due to the producer/consumer nature of the Oracle Server's query operations, only two operations in a given tree need to be performed simultaneously to minimize execution time.

To illustrate intra-operation parallelism and inter-operator parallelism, consider the following statement:

```
SELECT * FROM emp ORDER BY ename;
```

The execution plan consists of a full scan of the EMP table followed by a sorting of the retrieved rows based on the value of the ENAME column. For the sake of this example, assume the ENAME column is not indexed. Also assume that the degree of parallelism for the query is set to four, which means that four query servers can be active for any given operation. Figure 6 – 5 illustrates the parallel execution of our example query.

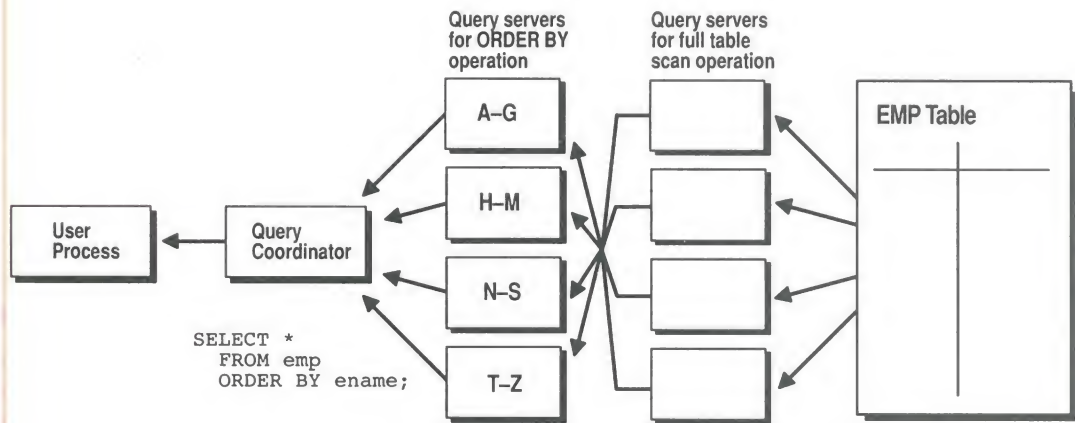


Figure 6 – 5 Inter-Operator Parallelism and Dynamic Partitioning

As you can see from Figure 6 – 5, there are actually eight query servers involved in the query even though the degree of parallelism is four. This is because a parent and child operator can be performed at the same time. Also note that all of the query servers involved in the scan operation send rows to the appropriate query server performing the sort operation. If a row scanned by a query server contains a value for the ENAME column between A and G, that row gets sent to the first ORDER BY query server. When the scan operation is complete, the sorting query servers can return the sorted results to the query coordinator, which in turn returns the complete query results to the user.

Note: When a set of query servers completes its operation, it moves on to operations higher in the data flow. For example, in the previous diagram, if there was another ORDER BY operation after the ORDER BY, the query servers performing the table scan perform the second ORDER BY operation after completing the table scan.

Setting the Degree of Parallelism

The query coordinator process may enlist two or more of the instance's query server processes to process the statement. The number of query server processes associated with a single operation is known as the *degree of parallelism*. The degree of parallelism is specified at the query level (with hints), at the table level (in the table's definition), or by default in the initialization parameter file. Note that the degree of parallelism applies only to the intra-operation parallelism. If inter-operation parallelism is possible, the total number of query servers can be twice the specified degree of parallelism.

Determining the Degree of Parallelism for Operations

The query coordinator determines the degree of parallelism by considering three specifications. The query coordinator first checks for query hints, then looks at the table's definition, and finally checks initialization parameters for the instance for the default degree of parallelism. Once a degree of parallelism is found in one of these specifications, it becomes the degree of parallelism for the query.

Note: The default degree of parallelism is used for tables that have PARALLEL attributed to them in the data dictionary, or via the PARALLEL hint. If a table has no parallelism attributed to it, or has NOPARALLEL (the default) attributed to it, then that table is never scanned in parallel—regardless of the default degree of parallelism that may be set by instance initialization parameters.

For queries involving more than one table, the query coordinator requests the greatest number specified for any table in the query. For example, on a query joining the EMP and DEPT tables, if EMP's degree of parallelism is specified as 5 and DEPT's degree of parallelism is specified as 6, the query coordinator would request six query servers for each operation in the query.

Keep in mind that no more than two operations can be performed simultaneously. Therefore, the maximum number of query servers requested for any query can be up to twice the degree of parallelism per instance.

Hints, the table definitions, or initialization parameters only determine the number of query servers that the query coordinator requests for a given operation. The actual number of query servers used depends upon how many query servers are available in the query server pool and whether inter-operation parallelism is possible.

When you create a table and populate it with a subquery in parallel, the degree of parallelism for the population is determined by the table's degree of parallelism. If no degree of parallelism is specified in the newly created table, the degree of parallelism is derived from the subquery's parallelism. If the subquery cannot be parallelized, the table is created serially.

Hints

Hints allow you to set the degree of parallelism for a table in a query and the caching behavior of the query. Refer to Chapter 7, "Tuning SQL Statements", for a general discussion on using hints in queries and the specific syntax for the PARALLEL, NOPARALLEL, CACHE, and NOCACHE hints.

Table and Cluster Definition Syntax

You can specify the degree of parallelism within a table definition. Use the CREATE TABLE, ALTER TABLE, CREATE CLUSTER, or ALTER CLUSTER statements to set the degree of parallelism for a table or clustered table. Refer to the *Oracle7 Server SQL Reference* for the complete syntax of those commands.

Default Degree of Parallelism

Oracle determines the number of disks that the table is stored on and the number of CPUs in the system and then selects the smaller of these two values as the default degree of parallelism. The default degree of parallelism is used when you do not specify a degree of parallelism in a hint or within a table's definition.

For example, your system has 20 CPUs and you issue a parallel query on a table that is stored on 15 disk drives. The default degree of parallelism for your query is 15 query servers.

Minimum Number of Query Servers

Oracle can perform a query in parallel as long as there are at least two query servers available. If there are too few query servers available, your query may execute slower than expected. You can specify that a minimum percentage of requested query servers must be available in order for the query to execute. This ensures that your query executes with a minimum acceptable parallel query performance. If the minimum percentage of requested servers are not available, the query does not execute and returns an error.

Specify the desired minimum percentage of requested query servers with the initialization parameter PARALLEL_QUERY_MIN_PERCENT. For example, if you specify 50 for this parameter, then at least 50% of the

query servers requested for any parallel operation must be available in order for the operation to succeed. If 20 query servers are requested, then at least 10 must be available or an error is returned to the user. If `PARALLEL_QUERY_MIN_PERCENT` is set to null, then all parallel operations will proceed as long as at least two query servers are available for processing.

Note: The parameters `PARALLEL_DEFAULT_SCANSIZE` and `PARALLEL_DEFAULT_MAX_SCANS` are obsolete in release 7.3.

Limiting the Number of Available Instances

The `INSTANCES` keyword of the `CREATE/ALTER TABLE/CLUSTER` commands allows you to specify that a table or cluster is split up among the buffer caches of all available instances of an Oracle Parallel Server. If you do not want tables to be dynamically partitioned among all the available instances, you can specify the number of instances that can participate in scanning or caching with the parameter `PARALLEL_DEFAULT_MAX_INSTANCES` or the `ALTER SYSTEM` command.

If you want to specify the number of instances to participate in parallel query processing at startup time, you can specify a value for the initialization parameter `PARALLEL_DEFAULT_MAX_INSTANCES`. See the *Oracle7 Server Reference* for more information about this parameter.

If you want to limit the number of instances available for parallel query processing dynamically, use the `ALTER SYSTEM` command. For example, if you have ten instances running in your Parallel Server, but you want only eight to be involved in parallel query processing, you can specify a value by issuing the following command:

```
ALTER SYSTEM SET SCAN_INSTANCES = 8;
```

Therefore, if a table's definition has a value of ten specified in the `INSTANCES` keyword, the table will be scanned by query servers on eight of the ten instances. Oracle selects the first eight instances in this example. Set the parameter `PARALLEL_MAX_SERVERS` to zero on the instances that you do not want to participate in parallel query processing.

If you wish to limit the number of instances that cache a table, you can issue the following command:

```
ALTER SYSTEM SET CACHE_INSTANCES = 8;
```

Therefore, if a table specifies the `CACHE` keyword with the `INSTANCES` keyword specified as 10, it will divide evenly among eight of the ten available instances' buffer caches.

Managing the Query Servers

When you start your instance, the Oracle Server creates a pool of query server processes available for any query coordinator. The number of query server processes that the Oracle Server creates at instance startup is specified by the initialization parameter `PARALLEL_MIN_SERVERS`.

Query server processes remain associated with a statement throughout its execution phase. When the statement is completely processed, its query server processes become available to process other statements. The query coordinator process returns any resulting data to the user process issuing the statement.

Varying Pool of Query Server Processes

If the volume of SQL statements processed concurrently by your instance changes drastically, the Oracle Server automatically changes the number of query server processes in the pool to accommodate this volume.

- If this volume increases, the Oracle Server automatically creates additional query server processes to handle incoming statements. The maximum number of query server processes for your instance is specified by the initialization parameter `PARALLEL_MAX_SERVERS`.
- If this volume subsequently decreases, the Oracle Server terminates a query server process if it has been idle for the period of time specified by the initialization parameter `PARALLEL_SERVER_IDLE_TIME`. The Oracle Server does not reduce the size of the pool below the value of `PARALLEL_MIN_SERVERS` regardless of how long the query server processes have been idle.

If all query servers in the pool are occupied and the maximum number of query servers has been started, a query coordinator processes the statement sequentially. "Tuning the Query Servers" on page 6 – 17 describes how to monitor an instance's pool of query servers and determine the appropriate values of the initialization parameters.

Tuning for the Parallel Query Option

Four questions are very important in tuning when using the parallel query option:

- what systems are appropriate for the parallel query option?
- how does I/O and file striping relate to the parallel query option?
- how should I set the degree of parallelism?
- how many query server processes do I need on my system?

This section details some of the factors to consider when answering the above questions. This section also details the EXPLAIN PLAN facility for examining a query's execution plan and parallelization scheme. Some factors involved in determining the answers to the questions above are operating system dependent.



Additional Information: See your operating system-specific Oracle documentation for more information about tuning while using the parallel query option.

What Systems Benefit? The parallel query option can help the most on the following systems:

- symmetric multiprocessors (SMP) or massively parallel systems
- systems with high I/O bandwidth (that is, many datafiles on many different disk drives)
- systems with underutilized or intermittently used CPUs (for example, systems where CPU usage is typically less than 30%)
- systems with sufficient memory to devote to sorting space for queries

If any one of these conditions is not true for your system, the parallel query option may not significantly help performance. In fact, on over-utilized systems or systems with small I/O bandwidth, the parallel query option can impede system performance.

It is important to realize that Oracle cannot return results to a user process in parallel. Therefore, if a query returns a large number of rows, the execution of the query will be faster, but the user process receives the rows serially. The parallel query option is best for queries that involve computing summary information for a large amount of data.

What Are the I/O Considerations?

The parallel query option works best on files that are spread or *striped* across many disk drives. This allows the query servers to maximize concurrent access to the datafiles.

Automatic file striping by the operating system is nearly as good as manually striping your tables across datafiles, so take advantage of automatic file striping if supported in your operating system. Operating system striping is usually easy to configure and provides automatic I/O balancing. Operating system striping is also optimal for random I/O operations such as sorts using temporary segments. Manual table striping is optimal for sequential I/O and may improve the performance for table scans.

File stripe size should be a few multiples larger than the size of `DB_FILE_MULTIBLOCK_READ_COUNT` blocks (the amount of blocks that Oracle attempts to read with a single I/O).



OSDoc

Additional Information: For a discussion of manually striping tables across datafiles, refer to “Striping Table Data” on page 9 – 5. For more information about automatic file striping and tools to use to determine I/O distribution among your devices, refer to your operating system documentation.

It is also important to note that large contiguous extents can help the query coordinator break up scan operations more efficiently for the query servers. During a scan operation, the query coordinator identifies all of the contiguous ranges of blocks and then breaks up these ranges into large, medium, and small groups of blocks. Each query server is given a large group of contiguous blocks to begin with and then is given successively smaller groups of blocks until the scan is completed. This ensures that all of the query servers complete the scan at approximately the same time. If there are several large extents in a table, the query coordinator can easily find groups of contiguous blocks to dispatch to the query servers.

The temporary tablespace used for sorting should also be striped across several disks (by the operating system, if possible). This tablespace should also contain several large extents.

How Should I Set the Degree of Parallelism?

After handling the I/O distribution, the degree of parallelism is the most important factor in tuning the parallel query option. You must first determine the maximum number of query servers your system can support and then divide those query servers among the estimated number of concurrent queries. The maximum number of query servers your system can support depends upon combinations of the following factors:

- the number and capacity of CPUs on your system
- the limitations on the number of processes on your system
- if tables are striped, the number of disk drives the table is striped across
- the amount of query processing on your system
- the location of the data (that is, whether it is cached or on disk)
- the type of operations in a query (for example, sorts or full table scans)

While some of these factors are operating system dependent, most of these factors are dependent upon the characteristics of your specific system and the type of operations involved in each query. Therefore, it is difficult to make generic recommendations for the degree of parallelism. It is often best to pick a degree of parallelism for a query, determine the execution time, and then determine the execution time of several higher and lower degrees of parallelism to find the optimal number.



OSDoc

Additional Information: To determine whether your system is being fully utilized, there are several graphical system monitors available on most operating systems. These monitors often give you a better idea of CPU utilization and system performance than monitoring the execution time of a query. Consult your operating system documentation to determine if your system supports graphical system monitors.

For single-user parallel execution, one guideline for your initial estimate of the degree of parallelism is to use one or two times the number of CPUs on your system (depending on the capacity of your CPUs). This is a good guess for sort-intensive operations because those tend to be CPU-bound. If your query is I/O intensive (that is, more scanning of tables than sorting), you may want to use one or two times the number of disk drives involved in the scans as your initial guess at the degree of parallelism.

If you have several concurrent queries that all require parallel execution, you should consider adding CPUs and/or disk drives to your system to allow for a greater number of query servers per query. If you cannot expand your system to accommodate multiple users, you may need to consider limiting parallel query execution to fewer users or lowering the degree of parallelism for each user's query.

Guidelines for Creating and Populating Tables and Indexes in Parallel

When you create a table and populate it with a subquery (CREATE TABLE...AS SELECT) in parallel, you want to evenly divide the population operation among all available processors. The goal is to distribute the I/O among the processors so that there is little contention for the I/O devices. A guideline for setting the degree of parallelism is two to four query servers per processor on your system.

You also want to avoid fragmentation by having as many files in your tablespace as the degree of parallelism for index or table creation. Each query server allocates an extent in a file when building a table or index. If more than one query server uses the same file, there may be gaps where one query server did not fill an extent completely. Try to have the degree of parallelism for parallel index and table creation come as close as possible to the number of files in the tablespace.

Memory Requirements for Sort Operations

The SORT_AREA_SIZE initialization parameter specifies the amount of memory to allocate per query server for sort operations. Setting this parameter to a larger value can dramatically increase the performance of sort operations since the entire sort is more likely to be performed in memory. Keep in mind that the actual amount of memory used for a sort is the value of SORT_AREA_SIZE multiplied by the number of query servers (for example, 10 query servers each with 1M for sorts requires 10M of memory allocated for the sort).

If memory is abundant on your system, you can benefit from setting SORT_AREA_SIZE to a large value. However, if memory is a concern for your system, you may want to limit the amount of memory allocated for sorts and increase the size of the buffer cache so that data blocks from temporary sort segments can be cached in the buffer cache.

Using Direct Disk I/O for Sorts with the Parallel Query Option

If memory and temporary space are abundant on your system, and you perform many large sorts to disk, you can set the initialization parameter SORT_DIRECT_WRITES to increase sort performance. Note that each query server will allocate its own set of direct write buffers (when SORT_DIRECT_WRITES is set to TRUE), so the total amount of memory allocated for buffers is the number of query servers multiplied by SORT_WRITE_BUFFERS, the number of direct write buffers, multiplied by SORT_WRITE_BUFFER_SIZE. In addition to the increased

Tuning the Query Servers

memory usage, sorts that use direct writes will tend to consume more temporary segment space on disk.

Setting SORT_DIRECT_WRITES to AUTO causes Oracle to allocated the memory for the direct write buffers out of the total sort area.

The V\$PQ_SYSSTAT view contains statistics that are useful for determining the appropriate number of query server processes for an instance. The statistics that are particularly useful are "Servers Busy", "Servers Idle", "Servers Started", and "Servers Shutdown".

Frequently, you will not be able to increase the maximum number of query servers for an instance because the maximum number is heavily dependent upon the capacity of your CPUs and your I/O bandwidth. However, if servers are continuously starting and shutting down, you should consider increasing the value of the parameter PARALLEL_MIN_SERVERS.

For example, if you have determined that the maximum number of concurrent query servers that your machine can manage is 100, you should set PARALLEL_MAX_SERVERS to 100. Next determine how many query servers the average query needs, and how many queries are likely to be executed concurrently. For this example, let's assume you'll have two concurrent queries with 20 as the average degree of parallelism. Thus, at any given point in time, there could be 80 query servers busy on an instance. Thus you should set the parameter PARALLEL_MIN_SERVERS to be 80.

Now you should periodically examine V\$PQ_SYSSTAT to determine if the 80 query servers for the instance are actually busy. To determine if the instance's query servers are active, issue the following query:

```
SELECT * FROM V$PQ_SYSSTAT
      WHERE statistic = "Servers Busy";
```

STATISTIC	VALUE
Servers Busy	70

If you find that there are typically fewer than PARALLEL_MIN_SERVERS busy at any given time, your idle query servers are additional system overhead that is not being used. You should then consider decreasing the value of the parameter PARALLEL_MIN_SERVERS. If you find that there are typically more query servers active than the value of PARALLEL_MIN_SERVERS and the "Servers Started" statistic is continuously growing, then you should consider increasing the value of the parameter PARALLEL_MIN_SERVERS.

EXPLAIN PLAN

The EXPLAIN PLAN facility details a query’s execution plan and how a query is parallelized. See Appendix A, “Performance Diagnostic Tools”, for a detailed description of the EXPLAIN PLAN facility.

The OBJECT_NODE and OTHER columns of the PLAN_TABLE describe the parallelism (if any) of a given query. For non–distributed queries, the OBJECT_NODE column describes the order in which the output from operations is consumed. The OTHER column describes the text of the query that is used by query servers for each operation. For example, in the following query:

```
EXPLAIN PLAN FOR
  SELECT dname, MAX(sal), AVG(sal)
  FROM emp, dept
  WHERE emp.deptno = dept.deptno
  GROUP BY dname;
```

the contents of the plan table may look something like this:

QUERY PLAN	OBJECT_NODE	OTHER
GROUP BY SORT	:Q704003	select c0, avg(c1), max(c1) from :Q704002 group by c0
MERGE JOIN	:Q704002	select /*+ ordered use_merge(a2) */ a2.c1 c0, a1.c1 c1 from :Q704001 a1, :Q704000 a2 where a1.c0 = a2.c0
JOIN SORT	:Q704002	operation combined with parent
FULL SCAN	:Q704001	select /*+ rowid(a1) */ a1.deptno c0, a1.sal c1 where rowid between :1 and :2
JOIN SORT	:Q704002	operation combined with parent
FULL SCAN	:Q704000	output consumed in parallel

From this output you can construct the data flow diagram for the query. Figure 6 – 6 illustrates the data flow diagram.

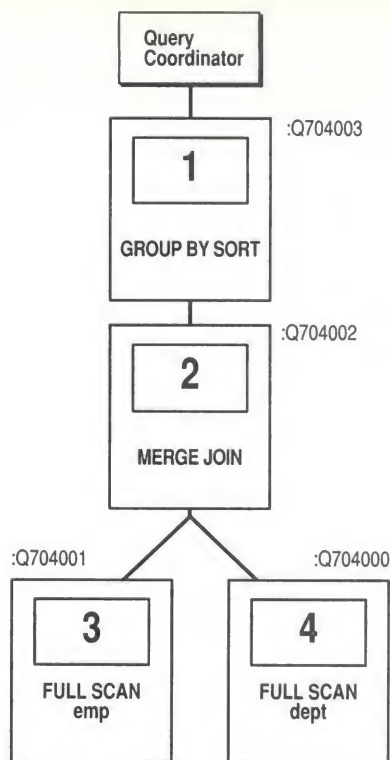


Figure 6 – 6 Data Flow Diagram with OBJECT_NODE Information

Combining Operations within a Tree

The EXPLAIN PLAN output for the preceding example join query describes that the MERGE JOIN operation also requires a JOIN SORT operation on both tables involved in the join. Because the MERGE JOIN operation requires the information from both JOIN SORT operations, the MERGE JOIN is considered the parent operation of the JOIN SORT. The query coordinator process compares the partitioning requirement of each operator with that of its parent.

- If they are compatible, the operators can be combined and performed together as a single operator.
- If they are incompatible, the query coordinator repartitions the rows returned by the operator so that the rows meet the partitioning requirements of its parent operator. The rows are then sent from the operator to its parent.

In the example above, the two JOIN SORT operations are compatible with their parent operation and therefore are combined into the MERGE JOIN. All other operations require communication of the results to the parent operation.

Parallel Index Creation

Multiple processes can work together simultaneously to create an index. By dividing the work necessary to create an index among multiple server processes, the Oracle Server can create the index more quickly than if a single server process created the index sequentially.

Parallel index creation works in much the same way as parallel execution of queries. One set of query servers scans the table to be indexed to obtain the rowids and column values for the rows. Then another set of query servers performs the sorting of the index entries based on the index column values and passes off the sorted entries to the coordinator process, which builds the B*-tree index from the sorted lists.

You can optionally specify that the creation of an index be unrecoverable. This means that no redo logging occurs during the index creation. This can significantly improve performance, but it sacrifices recoverability of the index in the event of media failure. If recoverability is not important to your application, you should consider using the UNRECOVERABLE option. For more information on recovery and the UNRECOVERABLE option, see the *Oracle7 Server Administrator's Guide*.

By default, the Oracle Server uses the table definition's PARALLEL clause value to determine the number of server processes to use when creating an index. You can override the default number of processes by using the PARALLEL clause in the CREATE INDEX command. Refer to the *Oracle7 Server SQL Reference* for the complete syntax of the CREATE INDEX command.



Attention: When creating an index in parallel, the STORAGE clause refers to the storage of each of the subindexes created by the query server processes. Therefore, an index created with an INITIAL of 5M and a PARALLEL DEGREE of 12 consumes at least 60M of storage during index creation because each process starts with an extent of 5M. When the query coordinator process combines the sorted subindexes, some of the extents may be trimmed, and the resulting index may be smaller than the requested 60M.

When you add or enable a UNIQUE key or PRIMARY KEY constraint on a table, you cannot automatically create the required index in parallel. Instead, manually create an index on the desired columns using the CREATE INDEX command and an appropriate PARALLEL clause and then add or enable the constraint. Oracle then uses the existing index when enabling or adding the constraint.

Refer to “Tuning for the Parallel Query Option” on page 6 – 13 for advice on tuning your system when creating indexes in parallel. For more information on how extents are allocated when using the parallel query option, see *Oracle7 Server Concepts*.

Tuning SQL Statements

Tuning your SQL statements is an important part of getting the best possible performance from Oracle. You should tune your SQL statements before your database administrator tunes Oracle itself:

- Even if you are not familiar with the internal workings of Oracle, you can significantly improve performance by tuning your application based on your knowledge of how Oracle executes SQL statements.
- If your SQL statements are not tuned well, they may not run well, even if Oracle is tuned well.

This chapter tells you

- how to write SQL statements for best performance of new applications
- how to use hints
- how to optimize the performance of SQL statements in existing applications
- how to use bitmap indexes

This chapter assumes that you are familiar with the concept of an execution plan and how the Oracle optimizer generates one. For this information, see Chapter 5, “The Optimizer”.

How to Write New SQL Statements

If you are writing SQL statements in a new application, follow these steps to optimize your statements:

- Create indexes that can be used by your statements.
- Create clusters to optimize your join statements.
- Create hash clusters that can be used by your statements.
- Choose an optimization approach for your statements.
- Use hints where appropriate in your statements.
- Compare alternative syntax for your statements.

This section discusses each of these steps.

How to Use Indexes

This section makes recommendations about creating indexes and discusses these issues:

- how to decide when to create indexes
- how to choose which columns to index
- how to use composite indexes
- how to write statements to use indexes

Once you have decided to create an index, you can create it with the CREATE INDEX command. For more information on creating indexes, see the *Oracle7 Server Application Developer's Guide*.

When to Create Indexes

Indexes improve the performance of queries that select a small percentage of rows from a table. As a general guideline, you should create indexes on tables that are often queried for less than 2% or 4% of the table's rows. This guideline is based on these assumptions:

- Rows with the same value for the column on which the query is based are uniformly distributed throughout the data blocks allocated to the table.
- Rows in the table are randomly ordered with respect to the column on which the query is based.
- Each data block allocated to the table contains at least 10 rows.
- The table contains a relatively small number of columns.
- Most queries on the table have relatively simple WHERE clauses.
- The cache hit ratio is low and there is no operating system cache.

If these assumptions do not describe the data in your table and the queries that access it, the percentage of the table's rows selected under which an index is helpful may increase to as much as 25%.

How to Choose Columns to Index

Follow these guidelines for choosing columns to index:

- Consider indexing columns that are used frequently in WHERE clauses.
- Consider indexing columns that are used frequently to join tables in SQL statements. For more information on optimizing joins, see the section "How to Use Clusters" on page 7-7.
- Only index columns with good selectivity. The *selectivity* of an index is the percentage of rows in a table having the same value for the indexed column. An index's selectivity is good if few rows have the same value.

Note: Oracle implicitly creates indexes on the columns of all unique and primary keys that you define with integrity constraints. These indexes are the most selective and the most effective in optimizing performance.

You can determine the selectivity of an index by dividing the number of rows in the table by the number of distinct indexed values. You can obtain these values using the ANALYZE command. A selectivity calculated in this manner should be interpreted as a percentage.

- Do not index columns with few distinct values. Such columns usually have poor selectivity and, therefore, do not optimize performance unless the frequently selected column values appear less frequently than the other column values.

For example, consider a column containing equal numbers of the values 'YES' and 'NO'. Indexing this column would not normally improve performance. However, if the value 'YES' appears relatively infrequently and your application often queries for 'YES', then indexing the column may improve performance.

- Do not index columns that are frequently modified. UPDATE statements that modify indexed columns and INSERT and DELETE statements that modify indexed tables take longer than if there were no index. Such SQL statements must modify data in indexes as well as data in tables.
- Do not index columns that only appear in WHERE clauses with functions or operators. A WHERE clause that uses a function (other than MIN or MAX) or an operator with an indexed column does not make available the access path that uses the index.

- Consider indexing foreign keys of referential integrity constraints in cases in which a large number of concurrent INSERT, UPDATE, and DELETE statements access the parent and child tables. Such an index allows Oracle to modify data in the child table without locking the parent table.

When choosing whether to index a column, consider whether the performance gain for queries is worth the performance loss for INSERT, UPDATE, and DELETE statements and the use of the space required to store the index. You may want to experiment and compare the processing times of your SQL statements with and without indexes. You can measure processing time with the SQL trace facility. For information on the SQL trace facility, see Appendix A, "Performance Diagnostic Tools".

How to Choose
Composite Indexes

A *composite index* is an index that is made up of more than one column. Composite indexes can provide additional advantages over single-column indexes:

better selectivity	Sometimes two or more columns, each with poor selectivity, can be combined in a composite index with good selectivity.
additional data storage	If all the columns selected by a query are in a composite index, Oracle can return these values from the index without accessing the table.

A SQL statement can use an access path involving a composite index if the statement contains constructs that use a leading portion of the index. A *leading portion* of an index is a set of one or more columns that were specified first and consecutively in the list of columns in the CREATE INDEX statement that created the index. Consider this CREATE INDEX statement:

```
CREATE INDEX comp_ind
ON tabl(x, y, z);
```

These combinations of columns are leading portions of the index: X, XY, and XYZ. These combinations of columns are not leading portions of the index: YZ and Z.

Follow these guidelines for choosing columns for composite indexes:

- Consider creating a composite index on columns that are frequently used together in WHERE clause conditions combined with AND operators, especially if their combined selectivity is better than the selectivity of either column individually.
- If several queries select the same set of columns based on one or more column values, consider creating a composite index containing all of these columns.

Of course, consider the guidelines associated with the general performance advantages and tradeoffs of indexes described in the previous sections. Follow these guidelines for ordering columns in composite indexes:

- Create the index so that the columns that are used in WHERE clauses make up a leading portion.
- If some of the columns are used in WHERE clauses more frequently, be sure to create the index so that the more frequently selected columns make up a leading portion to allow the statements that use only these columns to use the index.
- If all columns are used in WHERE clauses equally often, ordering these columns from most selective to least selective in the CREATE INDEX statement best improves query performance.
- If all columns are used in the WHERE clauses equally often but the data is physically ordered on one of the columns, place that column first in the composite index.

How to Write Statements That Use Indexes

After you create an index, the optimizer cannot use an access path that uses the index simply because the index exists. The optimizer can only choose such an access path for a SQL statement if it contains a construct that makes the access path available. For information on access paths and the constructs that make them available, see Chapter 5, “The Optimizer”.

To be sure that a SQL statement can use an access path that uses an index, be sure the statement contains a construct that makes such an access path available. If you are using the cost-based approach, you should also generate statistics for the index. Once you have made the access path available for the statement, the optimizer may or may not choose to use the access path, based on the availability of other access paths.

In some cases, you may want to prevent a SQL statement from using an access path that uses an existing index. You may want to do this if you know that the index is not very selective and that a full table scan would be more efficient. If the statement contains a construct that makes such an index access path available, you can force the optimizer to use a full table scan through one of these methods:

- You can make the index access path unavailable by modifying the statement in a way that does not change its meaning. The following example illustrates this method.
- You can use the FULL hint to force the optimizer to choose a full table scan instead of an index scan.
- You can use the INDEX or AND_EQUAL hint to force the optimizer to use one index or set of indexes instead of another.

Since the behavior of the optimizer may change in future versions of Oracle, relying on methods such as the first to choose access paths may not be a good long-range plan. Instead, use hints to suggest specific access paths to the optimizer. For information on hints, see the section “How to Use Hints” on page 7 – 14.

Example

Consider these queries that select rows from a table based on the value of a single column:

```
SELECT *  
  FROM tabl  
 WHERE coll = 'A'  
SELECT *  
  FROM tabl  
 WHERE coll = 'B';
```

Assume that the values of the COL1 column are the letters A through Z. Assume also that the table has 1000 rows and that 75% of those rows have a COL1 value of 'A'. Each of the other letters appears in 1% of the rows.

Since the value 'A' appears in 75% of the tables rows, the first query is likely to be executed faster with a full table scan than with an index scan using an index on the COL1 column. Since the value 'B' appears in 1% of the rows, an index scan is likely to be faster than a full table scan for the second query. For these reasons, it is desirable to create an index to be used by the second query, but it is not desirable to use this index for the first query. However, the number of occurrences of each distinct column value is not available to the optimizer. The optimizer is likely to choose the same access path for both of these queries, despite the disparity in the percentage of the table's rows each returns.

For the best performance of these queries, create an index on TAB1.COL1 so that it can be used by the second query:

```
CREATE INDEX coll_ind  
ON tab1(coll1);
```

Modify the WHERE clause of the first query so that it does not make available the access path that uses the COL1_IND index:

```
SELECT *  
FROM tab1  
WHERE coll1 || '' = 'A';
```

This change prevents the query from using the access path provided by COL1_IND. Index access paths are not available if the WHERE clause performs an operation or function on the indexed column. For this reason, the optimizer must choose a full table scan for this query.

Note: This change to the WHERE clause does not change the result of the condition, so it does not cause the query to return a different set of rows. For a column containing number or date data, you can achieve the same goal by modifying the WHERE clause condition so that the column value is added to 0.

How to Use Clusters

Follow these guidelines for choosing when to cluster tables:

- Consider clustering tables that are often accessed by your application in join statements.
- Do not cluster tables if your application only joins them occasionally or modifies their common column values frequently. Modifying a row's cluster key value takes longer than modifying the value in an unclustered table because Oracle may have to migrate the modified row to another block to maintain the cluster.
- Do not cluster tables if your application often performs full table scans of only one of the tables. A full table scan of a clustered table can take longer than a full table scan of an unclustered table. Oracle is likely to read more blocks since the tables are stored together.
- Consider clustering master-detail tables if you often select a master record and then the corresponding detail records. Since the detail records are stored in the same data block(s) as the master record, they are likely to still be in memory when you select them, so Oracle may perform less I/O.

- Consider storing a detail table alone in a cluster if you often select many detail records of the same master. This measure improves the performance of queries that select detail records of the same master but does not decrease the performance of a full table scan on the master table.
- Do not cluster tables if the data from all tables with the same cluster key value exceeds more than one or two Oracle blocks. To access a row in a clustered table, Oracle reads all blocks containing rows with that value. If these rows take up multiple blocks, accessing a single row could require more reads than accessing the same row in an unclustered table.

Consider the benefits and drawbacks of clusters with respect to the needs of your application. For example, you may decide that the performance gain for join statements outweighs the performance loss for statements that modify cluster key values. You may want to experiment and compare processing times with your tables both clustered and stored separately. To create a cluster, use the `CREATE CLUSTER` command. For more information on creating clusters, see the *Oracle7 Server Application Developer's Guide*.

How to Use Hashing

Follow these guidelines for choosing when to use hash clusters:

- Consider using hash clusters to store tables that are often accessed by SQL statements with `WHERE` clauses that contain equality conditions that use the same column or combination of columns. Designate this column or combination of columns as the cluster key.
- Store a table in a hash cluster if you can determine how much space is required to hold all rows with a given cluster key value, including rows to be inserted immediately as well as rows to be inserted in the future.
- Do not use hash clusters if space in your database is scarce and you cannot afford to allocate additional space for rows to be inserted in the future.
- Do not use a hash cluster to store a constantly growing table if the process of occasionally creating a new, larger hash cluster to hold that table is impractical.

- Do not store a table in a hash cluster if your application often performs full scans of the table and you feel you must allocate a great deal of extra space to the hash cluster in anticipation of the table growing a great deal in the future. Such full table scans must read all blocks allocated to the hash cluster, even though some blocks may contain few rows. Storing the table alone would reduce the number of blocks read by full table scans.
- Do not store a table in a hash cluster if your application frequently modifies the cluster key values. Modifying a row's cluster key value can take longer than modifying the value in an unclustered table because Oracle may have to migrate the modified row to another block to maintain the cluster.
- Storing a single table in a hash cluster can be useful, regardless of whether the table is often joined with other tables, provided that hashing is appropriate for the table based on the previous points in this list.

Consider the benefits and drawbacks of hash clusters with respect to the needs of your application. You may want to experiment and compare processing times with a table both stored in a hash cluster and stored alone with an index. To create a hash cluster, use the `CREATE CLUSTER` command with the `HASH` and `HASHKEYS` parameters. For more information on creating hash clusters, see the *Oracle7 Server Application Developer's Guide*.

How to Determine How Many Hash Values to Use

When you create a hash cluster, you must use the `HASHKEYS` parameter of the `CREATE CLUSTER` statement to specify the number of hash values for the hash cluster. For best performance of hash scans, choose a `HASHKEYS` value that is at least as large as the number of cluster key values. Such a value reduces the chance of *collisions*, or multiple cluster key values resulting in the same hash value. Collisions force Oracle to test the rows in each block for the correct cluster key value after performing a hash scan. Collisions reduce the performance of hash scans.

Oracle always rounds up the `HASHKEYS` value that you specify to the nearest prime number to obtain the actual number of hash values. This rounding is designed to reduce collisions.

How to Use Anti_Joins

An anti-join is a form of join with reverse logic. Instead of returning rows when there is a match (according to the join predicate) between the left and right side, an anti-join will return those rows from the left side of the predicate for where there was no match on the right. This behavior is exactly that of a `NOT IN` subquery with the right side of the anti-join predicate corresponding to the subquery.

An anti-join uses sort-merge or hash joins to evaluate the NOT IN subquery provided that certain conditions are met. Assume that the subquery predicate is of the form (colA1, colA2, ... colAn) NOT IN (SELECT colB1, colB2, ..., colBn FROM ...). The following conditions must be true for the subquery to be transformed into a hash or sort-merge anti-join:

- All of the references to the columns in A must be simple reference to columns. References to columns in B must either be simple references to columns or aggregate functions (MIN, MAX, SUM, COUNT, or AVG) applied directly to a simple column if the subquery contains a GROUP BY. No other expressions are allowed.
- All column references must be known to be not null.
- The subquery must not have any correlation predicates. That is, predicates referencing anything in surrounding query blocks.
- The WHERE clause of the surrounding query must not have ORs at the top-most logical level.
- Anti-joins can only be used with the cost-based approach.

Oracle transforms NOT IN subqueries into sort-merge or hash anti-joins if the conditions in the previous section are true and there is a hint or initialization parameter specifying that the transformation take place.

For a specific query, place the MERGE_AJ or HASH_AJ hints into the NOT IN subquery. MERGE_AJ uses a sort-merge anti-join and HASH_AJ uses a hash anti-join. For example:

```
SELECT * FROM emp
  WHERE ename LIKE 'J%' AND
         deptno IS NOT NULL AND
         deptno NOT IN (SELECT /*+ HASH_AJ */ deptno FROM dept
                        WHERE deptno IS NOT NULL AND
                               loc = 'DALLAS');
```

If you wish the anti-join transformation to always occur if the conditions in the previous section are met, set the ALWAYS_ANTI_JOIN initialization parameter to MERGE or HASH. The transformation to the corresponding anti-join type will always take place when possible.

How to Choose an Optimization Approach

This section discusses

- when to use the cost-based approach
- how to choose a goal for the cost-based approach
- when and how to generate statistics for the cost-based approach
- when to use the rule-based approach

When to Use the Cost-Based Approach

In general, you should use the cost-based approach for all new applications. The cost-based approach generally chooses an execution plan that is as good as or better than the plan chosen by the rule-based approach, especially for large queries with multiple joins or multiple indexes. The cost-based approach also improves productivity by eliminating the need for you to tune your SQL statements yourself.

To enable cost-based optimization for a statement, collect statistics for the tables accessed by the statement and be sure the `OPTIMIZER_MODE` initialization parameter is set to its default value of `CHOOSE`.

You can also enable cost-based optimization in these ways:

- To enable cost-based optimization for your session only, issue an `ALTER SESSION` statement with an `OPTIMIZER_GOAL` option value of `ALL_ROWS` or `FIRST_ROWS`.
- To enable cost-based optimization for an individual SQL statement, use the `ALL_ROWS` or `FIRST_ROWS` hint. For information on hints, see the section “How to Use Hints” on page 7 – 14.

Generating Statistics Since the cost-based approach relies on statistics, you should generate statistics for all tables, clusters, and indexes accessed by your SQL statements before using the cost-based approach. If the size and data distribution of these tables changes frequently, you should generate these statistics regularly to ensure that they accurately represent the data in the tables.

Oracle can generate statistics using these techniques:

- estimation based on random data sampling
- exact computation

Use estimation, rather than computation, unless you think you need exact values:

- Computation always provides exact values, but can take longer than estimation. The time necessary to compute statistics for a table is approximately the time required to perform a full table scan and a sort of the rows of the table.
- Estimation is often much faster than computation, especially for large tables, because estimation never scans the entire table.

To perform a computation, Oracle requires enough space to perform a scan and sort of the table. If there is not enough space in memory, temporary space may be required. For estimations, Oracle requires enough space to perform a scan and sort of all of the rows in the requested sample of the table.

Because of the time and space required for the computation of table statistics, it is usually best to perform an estimation with a 20% sample size for tables and clusters. For indexes, computation does not take up as much time or space, so it is best to perform a computation.

When you generate statistics for a table, column, or index, if the data dictionary already contains statistics for the analyzed object, Oracle updates the existing statistics with the new ones. Oracle invalidates any currently parsed SQL statements that access any of the analyzed objects. When such a statement is next executed, the optimizer automatically chooses a new execution plan based on the new statistics. Distributed statements issued on remote databases that access the analyzed objects use the new statistics when they are next parsed.

Some statistics are always computed, regardless of whether you specify computation or estimation. If you choose estimation and the time saved by estimating a statistic is negligible, Oracle computes the statistic.

You can generate statistics with the ANALYZE command.

Example This example generates statistics for the EMP table and its indexes:

```
ANALYZE TABLE emp  
ESTIMATE STATISTICS;
```

Choosing a Goal for the Cost-Based Approach The execution plan produced by the optimizer can vary depending upon the optimizer's goal. Optimizing for best throughput is more likely to result in a full table scan rather than an indexed scan or a sort-merge join rather than a nested loops join. Optimizing for best response time is more likely to result in an index scan or a nested loops join.

For example, consider a join statement that can be executed with either a nested loops operation or a sort-merge operation. The sort-merge operation may return the entire query result faster, while the nested loops operation may return the first row faster. If the goal is best throughput, the optimizer is more likely to choose a sort-merge join. If the goal is best response time, the optimizer is more likely to choose a nested loops join.

Choose a goal for the optimizer based on the needs of your application:

- For applications performed in batch, such as Oracle Reports applications, optimize for best throughput. Throughput is usually more important in batch applications because the user initiating the application is only concerned with the time necessary for the application to complete. Response time is less important because the user does not examine the results of individual statements while the application is running.
- For interactive applications, such as Oracle Forms applications or SQL*Plus queries, optimize for best response time. Response time is usually important in interactive applications because the interactive user is waiting to see the first row accessed by the statement.
- For queries that use ROWNUM to limit the number of rows, optimize for best response time. Because of the semantics of ROWNUM queries, optimizing for response time provides the best results.

By default, the cost-based approach optimizes for best throughput. You can change the goal of the cost-based approach in these ways:

- To change the goal of the cost-based approach for all SQL statements in your session, issue an ALTER SESSION statement with the OPTIMIZER_GOAL option.
- To specify the goal of the cost-based approach for an individual SQL statement, use the ALL_ROWS or FIRST_ROWS hint. For information on hints, see the section “How to Use Hints” on page 7 – 14.

Example This statement changes the goal of the cost-based approach for your session to best response time:

```
ALTER SESSION
  SET OPTIMIZER_GOAL = FIRST_ROWS;
```

When to Use Rule-Based Optimization

If you have developed applications using a previous version of Oracle and have carefully tuned your SQL statements based on the rules of the optimizer, you may want to continue using rule-based optimization when you upgrade these applications to Oracle7.

If you neither collect statistics nor add hints to your SQL statements, your statements will continue to use rule-based optimization. However, you should eventually migrate your existing applications to use the cost-based approach, because the rule-based approach will not be available in future versions of Oracle.

You can enable cost-based optimization on a trial basis simply by collecting statistics. You can then return to rule-based optimization by deleting them or by setting either the value of the `OPTIMIZER_MODE` initialization parameter or the `OPTIMIZER_GOAL` parameter of the `ALTER SESSION` command to `RULE`. You can also use this value if you want to collect and examine statistics for your data without using the cost-based approach.

How to Use Hints

As an application designer, you may know information about your data that the optimizer cannot. For example, you may know that a certain index is more selective for certain queries than the optimizer can determine. Based on this information, you may be able to choose a more efficient execution plan than the optimizer can. In such a case, you can use hints to force the optimizer to use your chosen execution plan.

Hints are suggestions that you give the optimizer for optimizing a SQL statement. Hints allow you to make decisions usually made by the optimizer. You can use hints to specify

- the optimization approach for a SQL statement
- the goal of the cost-based approach for a SQL statement
- the access path for a table accessed by the statement
- the join order for a join statement
- a join operation in a join statement

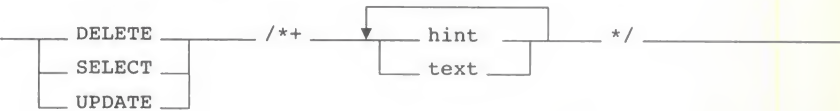
Hints apply only to the optimization of the statement block in which they appear. A *statement block* is any one of the following statements or parts of statements:

- a simple `SELECT`, `UPDATE`, or `DELETE` statement
- a parent statement or subquery of a complex statement
- a part of a compound query

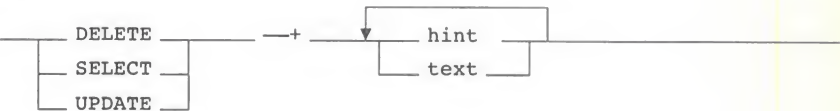
For example, a compound query consisting of two component queries combined by the UNION operator has two statement blocks, one for each component query. For this reason, hints in this first component query apply only to its optimization, not to the optimization of the second component query.

You can send hints for a SQL statement to the optimizer by enclosing them in a comment within the statement. For more information on comments, see Chapter 2, "Elements of SQL", of the *Oracle7 Server SQL Reference*.

A statement block can have only one comment containing hints. This comment can only follow the SELECT, UPDATE, or DELETE keyword. The syntax diagrams show the syntax for hints contained in both styles of comments that Oracle supports within a statement block.



or



where:

DELETE SELECT UPDATE	Is a DELETE, SELECT, or UPDATE keyword that begins a statement block. Comments containing hints can only appear after these keywords.
+	Is a plus sign that causes Oracle to interpret the comment as a list of hints. The plus sign must follow immediately after the comment delimiter (no space is permitted).
hint	Is one of the hints discussed in this section. If the comment contains multiple hints, each pair of hints must be separated by at least one space.
text	Is other commenting text that can be interspersed with the hints.

If you specify hints incorrectly, Oracle ignores them, but does not return an error:

- Oracle ignores hints if the comment containing them does not follow a DELETE, SELECT, or UPDATE keyword.
- Oracle ignores hints containing syntax errors, but considers other correctly specified hints within the same comment.
- Oracle ignores combinations of conflicting hints, but considers other hints within the same comment.

Oracle also ignores hints in all SQL statements in environments that use PL/SQL Version 1, such as SQL*Forms Version 3 triggers.

The optimizer only recognizes hints when using the cost-based approach. If you include any hint (except the RULE hint) in a statement block, the optimizer automatically uses the cost-based approach.

The following sections show the syntax of each hint.

Hints for Optimization Approaches and Goals

The hints described in this section allow you to choose between the cost-based and the rule-based optimization approaches and, with the cost-based approach, between the goals of best throughput and best response time. If a SQL statement contains a hint that specifies an optimization approach and goal, the optimizer uses the specified approach regardless of the presence or absence of statistics, the value of the OPTIMIZER_MODE initialization parameter, and the OPTIMIZER_GOAL parameter of the ALTER SESSION command.

Note: The optimizer goal applies only to queries submitted directly. Use hints to determine the access path for SQL statements submitted from within PL/SQL. The ALTER SESSION OPTIMIZER_GOAL statement does not affect SQL that is run from within PL/SQL.

ALL_ROWS

The ALL_ROWS hint explicitly chooses the cost-based approach to optimize a statement block with a goal of best throughput (that is, minimum total resource consumption). For example, the optimizer uses the cost-based approach to optimize this statement for best throughput:

```
SELECT /*+ ALL_ROWS */ empno, ename, sal, job
  FROM emp
 WHERE empno = 7566;
```

FIRST_ROWS

The FIRST_ROWS hint explicitly chooses the cost-based approach to optimize a statement block with a goal of best response time (minimum resource usage to return first row). This hint causes the optimizer to make these choices:

- If an index scan is available, the optimizer may choose it over a full table scan.
- If an index scan is available, the optimizer may choose a nested loops join over a sort–merge join whenever the associated table is the potential inner table of the nested loops.
- If an index scan is made available by an ORDER BY clause, the optimizer may choose it to avoid a sort operation.

For example, the optimizer uses the cost–based approach to optimize this statement for best response time:

```
SELECT /*+ FIRST_ROWS */ empno, ename, sal, job
  FROM emp
 WHERE empno = 7566;
```

The optimizer ignores this hint in DELETE and UPDATE statement blocks and in SELECT statement blocks that contain any of the following syntax:

- set operators (UNION, INTERSECT, MINUS, UNION ALL)
- GROUP BY clause
- FOR UPDATE clause
- group functions
- DISTINCT operator

These statements cannot be optimized for best response time because Oracle must retrieve all rows accessed by the statement before returning the first row. If you specify this hint in any of these statements, the optimizer uses the cost–based approach and optimizes for best throughput.

If you specify either the ALL_ROWS or FIRST_ROWS hint in a SQL statement and the data dictionary contains no statistics about any of the tables accessed by the statement, the optimizer uses default statistical values (such as allocated storage for such tables) to estimate the missing statistics and subsequently to choose an execution plan. Since these estimates may not be as accurate as those generated by the ANALYZE command, you should use the ANALYZE command to generate statistics for all tables accessed by statements that use cost–based optimization.

If you specify hints for access paths or join operations along with either the ALL_ROWS or FIRST_ROWS hint, the optimizer gives precedence to the access paths and join operations specified by the hints.

CHOOSE The CHOOSE hint causes the optimizer to choose between the rule-based approach and the cost-based approach for a SQL statement based on the presence of statistics for the tables accessed by the statement. If the data dictionary contains statistics for at least one of these tables, the optimizer uses the cost-based approach and optimizes with the goal of best throughput. If the data dictionary contains no statistics for any of these tables, the optimizer uses the rule-based approach.

In the following statement, if statistics are present for the EMP table, the optimizer uses the cost-based approach. If no statistics for the EMP table exist in the data dictionary, the optimizer uses the rule-based approach.

```
SELECT /*+ CHOOSE */
empno, ename, sal, job
  FROM emp
 WHERE empno = 7566;
```

RULE The RULE hint explicitly chooses rule-based optimization for a statement block. This hint also causes the optimizer to ignore any other hints specified for the statement block. For example, the optimizer uses the rule-based approach for this statement:

```
SELECT                                —+ RULE
empno, ename, sal, job
  FROM emp
 WHERE empno = 7566;
```

The RULE hint, along with the rule-based approach, will not be available in future versions of Oracle.

Hints for Access Methods Each hint described in this section suggests an access method for a table. Specifying one of these hints causes the optimizer to choose the specified access path only if the access path is available based on the existence of an index or cluster and the syntactic constructs of the SQL statement. For a discussion of access methods and a list of constructs and the access paths they make available, see Chapter 5, “The Optimizer”. If a hint specifies an unavailable access path, the optimizer ignores it.

You must specify the table to be accessed exactly as it appears in the statement. If the statement uses an alias for the table, you must use the alias, rather than the table name, in the hint. The table name within the hint should not include the schema name, if the schema name is present in the statement.

FULL The FULL hint explicitly chooses a full table scan for the specified table. The syntax of the FULL hint is

`FULL(table)`

where *table* specifies the name or alias of the table on which the full table scan is to be performed.

For example, Oracle performs a full table scan on the ACCOUNTS table to execute this statement, even if there is an index on the ACCNO column that is made available by the condition in the WHERE clause:

```
SELECT /*+ FULL(a) Don't use the index on ACCNO */ accno, bal
  FROM accounts a
 WHERE accno = 7086854;
```

Note: Because the ACCOUNTS table has an alias, A, the hint must refer to the table by its alias, rather than by its name. Also, do not specify schema names in the hint, even if they are specified in the FROM clause.

ROWID The ROWID hint explicitly chooses a table scan by ROWID for the specified table. The syntax of the ROWID hint is

`ROWID(table)`

where *table* specifies the name or alias of the table on which the table access by ROWID is to be performed.

CLUSTER The CLUSTER hint explicitly chooses a cluster scan to access the specified table. The syntax of the CLUSTER hint is

`CLUSTER(table)`

where *table* specifies the name or alias of the table to be accessed by a cluster scan.

The following example illustrates the use of the CLUSTER hint.

```
SELECT —+ CLUSTER emp
  ename, deptno
  FROM emp, dept
 WHERE deptno = 10 AND
        emp.deptno = dept.deptno;
```

HASH The HASH hint explicitly chooses a hash scan to access the specified table. The syntax of the HASH hint is

`HASH(table)`

where *table* specifies the name or alias of the table to be accessed by a hash scan.

INDEX The INDEX hint explicitly chooses an index scan for the specified table. The syntax of the INDEX hint is



where:

<i>table</i>	Specifies the name or alias of the table associated with the index to be scanned.
<i>index</i>	Specifies an index on which an index scan is to be performed.

This hint may optionally specify one or more indexes:

- If this hint specifies a single available index, the optimizer performs a scan on this index. The optimizer does not consider a full table scan or a scan on another index on the table.
- If this hint specifies a list of available indexes, the optimizer considers the cost of a scan on each index in the list and then performs the index scan with the lowest cost. The optimizer may also choose to scan multiple indexes from this list and merge the results, if such an access path has the lowest cost. The optimizer does not consider a full table scan or a scan on an index not listed in the hint.
- If this hint specifies no indexes, the optimizer considers the cost of a scan on each available index on the table and then performs the index scan with the lowest cost. The optimizer may also choose to scan multiple indexes and merge the results, if such an access path has the lowest cost. The optimizer does not consider a full table scan.

For example, consider this query, which selects the name, height, and weight of all male patients in a hospital:

```
SELECT name, height, weight
FROM patients
WHERE sex = 'M';
```

Assume that there is an index on the SEX column and that this column contains the values M and F. If there are equal numbers of male and female patients in the hospital, the query returns a relatively large percentage of the table's rows and a full table scan is likely to be faster than an index scan. However, if a very small percentage of the hospital's patients are male, the query returns a relatively small percentage of the table's rows and an index scan is likely to be faster than a full table scan.

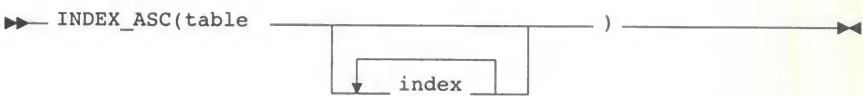
The number of occurrences of each distinct column value is not available to the optimizer. The cost-based approach assumes that each value has an equal probability of appearing in each row. For a column having only two distinct values, the optimizer assumes each value appears in 50% of the rows, so the cost-based approach is likely to choose a full table scan rather than an index scan.

If you know that the value in the WHERE clause of your query appears in a very small percentage of the rows, you can use the INDEX hint to force the optimizer to choose an index scan. In this statement, the INDEX hint explicitly chooses an index scan on the SEX_INDEX, the index on the SEX column:

```
SELECT /*+ INDEX(patients sex_index) Use SEX_INDEX, since there
                                are few male patients */
name, height, weight
FROM patients
WHERE sex = 'M';
```

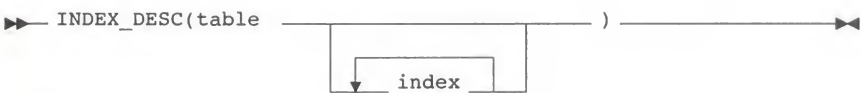
INDEX_ASC

The INDEX_ASC hint explicitly chooses an index scan for the specified table. If the statement uses an index range scan, Oracle scans the index entries in ascending order of their indexed values. The syntax of the INDEX_ASC hint is



Each parameter serves the same purpose as in the INDEX hint. Because Oracle’s default behavior for a range scan is to scan index entries in ascending order of their indexed values, this hint does not currently specify anything more than the INDEX hint. However, since Oracle Corporation does not guarantee that the default behavior for an index range scan will remain the same in future versions of Oracle, you may want to use the INDEX_ASC hint to specify ascending range scans explicitly, should the default behavior change.

INDEX_DESC The INDEX_DESC hint explicitly chooses an index scan for the specified table. If the statement uses an index range scan, Oracle scans the index entries in descending order of their indexed values. The syntax of the INDEX_DESC is



Each parameter serves the same purpose as in the INDEX hint. This hint has no effect on SQL statements that access more than one table. Such statements always perform range scans in ascending order of the indexed values. For example, consider this table, which contains the temperature readings of a tank of water holding marine life:

```
CREATE TABLE tank_readings
  (time          DATE      CONSTRAINT un_time UNIQUE,
   temperature   NUMBER );
```

Each of the table's rows stores a time and the temperature measured at that time. A UNIQUE constraint on the TIME column ensures that the table does not contain more than one reading for the same time.

Oracle enforces this constraint with an index on the TIME column. Consider this complex query, which selects the most recent temperature reading taken as of a particular time T. The subquery returns either T or the latest time before T at which a temperature reading was taken. The parent query then finds the temperature taken at that time:

```
SELECT temperature
  FROM tank_readings
 WHERE time = (SELECT MAX(time)
               FROM tank_readings
               WHERE time <= TO_DATE(:t) );
```

The execution plan for this statement looks like the following figure:



Figure 7 – 1 Execution Plan without Hints

To execute this statement, Oracle performs these operations:

- Steps 4 and 3 execute the subquery:
 - Step 4 performs a range scan of the UN_TIME index to return all the TIME values less than or equal to T.
 - Step 3 chooses the greatest TIME value from Step 4 and returns it.
- Steps 2 and 1 execute the parent query:
 - Step 2 performs a unique scan of the UN_TIME index based on the TIME value returned by Step 3 and returns the associated ROWID.
 - Step 1 accesses the TANK_READINGS table using the ROWID returned by Step 2 and returns the TEMPERATURE value.

In Step 4, Oracle scans the TIME values in the index in ascending order beginning with the smallest. Oracle stops scanning at the first TIME value greater than T and then returns all the values less than or equal to T to Step 3. Note that Step 3 needs only the greatest of these values. Using the INDEX_DESC hint, you can write an equivalent query that reads only one TIME value from the index:

```
SELECT /** INDEX_DESC(tank_readings un_time) */ temperature
FROM tank_readings
WHERE time <= TO_DATE(:t)
AND ROWNUM = 1
ORDER BY time DESC;
```

The execution plan for this query looks like the following figure:



Figure 7 – 2 Execution Plan while Using the INDEX_DESC Hint

To execute this statement, Oracle performs these operations:

- Step 3 performs a range scan of the UN_TIME index searching for TIME values less than or equal to T and returns their associated ROWIDs.
- Step 2 accesses the TANK_READINGS table by the ROWIDs returned by Step 3.
- Step 1 enforces the ROWNUM=1 condition by requesting only one row from Step 2.

Because of the INDEX_DESC hint, Step 3 scans the TIME values in the index in descending order beginning at T. The first TIME value scanned is either T (if the temperature was taken at T) or the greatest TIME value less than T. Since Step 1 requests only one row, Step 3 scans no more index entries after the first.

Since the default behavior is an ascending index scan, issuing this query without the INDEX_DESC hint would cause Oracle to begin scanning at the earliest time in the table, rather than at the latest time less than or equal to T. Step 1 would then return the temperature at the earliest time. You must use this hint to make this query return the same temperature as the complex query described earlier in this section.

AND_EQUAL The AND_EQUAL hint explicitly chooses an execution plan that uses an access path that merges the scans on several single-column indexes. The syntax of the AND_EQUAL hint is:

```
►► AND_EQUAL(table index index index index index ) ►►
```

where:

- table* Specifies the name or alias of the table associated with the indexes to be merged.
- index* Specifies an index on which an index scan is to be performed. You must specify at least two indexes. You cannot specify more than five.

USE_CONCAT The USE_CONCAT hint forces combined OR conditions in the WHERE clause of a query to be transformed into a compound query using the UNION ALL set operator. Normally, this transformation occurs only if the cost of the query using the concatenations is cheaper than the cost without them.

Hint for Join Orders The ORDERED hint suggests a join order.

ORDERED The ORDERED hint causes Oracle to join tables in the order in which they appear in the FROM clause. For example, this statement joins table TAB1 to table TAB2 and then joins the result to table TAB3:

```
SELECT /*+ ORDERED */ tab1.col1, tab2.col2, tab3.col3
FROM tab1, tab2, tab3
WHERE tab1.col1 = tab2.col1
      AND tab2.col1 = tab3.col1;
```

If you omit the ORDERED hint from a SQL statement performing a join, the optimizer chooses the order in which to join the tables.

You may want to use the ORDERED hint to specify a join order if you know something about the number of rows selected from each table that the optimizer does not. Such information would allow you to choose an inner and outer table better than the optimizer could.

Hints for Join Operations

Each hint described in this section suggests a join operation for a table. You must specify a table to be joined exactly as it appears in the statement. If the statement uses an alias for the table, you must use the alias rather than the table name in the hint. The table name within the hint should not include the schema name, if the schema name is present in the statement.

The USE_NL and USE_MERGE hints must be used with the ORDERED hint. Oracle uses these hints when the referenced table is forced to be the inner table of a join, and they are ignored if the referenced table is the outer table.

USE_NL The USE_NL hint causes Oracle to join each specified table to another row source with a nested loops join using the specified table as the inner table. The syntax of the USE_NL hint is



where *table* is the name or alias of a table to be used as the inner table of a nested loops join.

For example, consider this statement, which joins the ACCOUNTS and CUSTOMERS tables. Assume that these tables are not stored together in a cluster:

```
SELECT accounts.balance, customers.last_name, customers.first_name
  FROM accounts, customers
 WHERE accounts.custno = customers.custno;
```

Since the default goal of the cost-based approach is best throughput, the optimizer will choose either a nested loops operation or a sort-merge operation to join these tables, depending on which is likely to return all the rows selected by the query more quickly.

However, you may want to optimize the statement for best response time, or the minimal elapsed time necessary to return the first row selected by the query, rather than best throughput. If so, you can force the optimizer to choose a nested loops join by using the USE_NL hint. In

this statement, the `USE_NL` hint explicitly chooses a nested loops join with the `CUSTOMERS` table as the inner table:

```
SELECT /*+ ORDERED USE_NL(customers) Use N-L to get first row
        faster */
accounts.balance, customers.last_name, customers.first_name
FROM accounts, customers
WHERE accounts.custno = customers.custno;
```

In many cases, a nested loops join returns the first row faster than a sort-merge join. A nested loops join can return the first row after reading the first selected row from one table and the first matching row from the other and combining them, while a sort-merge join cannot return the first row until after reading and sorting all selected rows of both tables and then combining the first rows of each sorted row source.

`USE_HASH`

The `USE_HASH` hint causes Oracle to join each specified table with another row source with a hash join. The syntax of this hint is

►► `USE_HASH` (`table`) ►►

where *table* is a table to be joined to the row source resulting from joining the previous tables in the join order using a hash join.

`USE_MERGE`

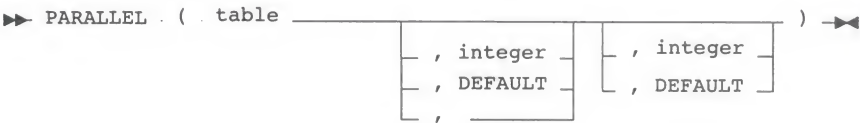
The `USE_MERGE` hint causes Oracle to join each specified table with another row source with a sort-merge join. The syntax of the `USE_MERGE` hint is

►► `USE_MERGE` (`table`) ►►

where *table* is a table to be joined to the row source resulting from joining the previous tables in the join order using a sort-merge join.

Each hint described in this section determines how statements are parallelized or not parallelized when using the parallel query option. Refer to Chapter 6, "Parallel Query Option", for more information on the parallel query option.

PARALLEL The PARALLEL hint allows you to specify the desired number of concurrent query servers that can be used for the query. The syntax is



The PARALLEL hint must use the table alias if an alias is specified in the query. The PARALLEL hint can then take two values separated by commas after the table name. The first value specifies the degree of parallelism for the given table, the second value specifies how the table is to be split among the instances of a Parallel Server. Specifying DEFAULT or no value signifies the query coordinator should examine the settings of the initialization parameters (described in a later section) to determine the default degree of parallelism.

In the following example, the PARALLEL hint overrides the degree of parallelism specified in the EMP table definition:

```
SELECT /*+ FULL(scott_emp) PARALLEL(scott_emp, 5) */
       ename
FROM   scott.emp scott_emp;
```

In the next example, the PARALLEL hint overrides the degree of parallelism specified in the EMP table definition and tells the optimizer to use the default degree of parallelism determined by the initialization parameters. This hint also specifies that the table should be split among all of the available instances, with the default degree of parallelism on each instance.

```
SELECT /*+ FULL(scott_emp) PARALLEL(scott_emp, DEFAULT,DEFAULT) */
       ename
FROM   scott.emp scott_emp;
```

NOPARALLEL The NOPARALLEL hint allows you to disable parallel scanning of a table, even if the table was created with a PARALLEL clause. The following example illustrates the NOPARALLEL hint:

```
SELECT /*+ NOPARALLEL(scott_emp) */
       ename
FROM   scott.emp scott_emp;
```

The NOPARALLEL hint is equivalent to specifying the hint
`/*+ PARALLEL(table,1,1) */.`

CACHE The CACHE hint specifies that the blocks retrieved for the table in the hint are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This option is useful for small lookup tables. In the following example, the CACHE hint overrides the table's default caching specification:

```
SELECT /*+ FULL (scott_emp) CACHE(scott_emp) */
       ename
FROM scott.emp scott_emp;
```

NOCACHE The NOCACHE hint specifies that the blocks retrieved for this table are placed at the least recently used end of the LRU list in the buffer cache when a full table scan is performed. This is the normal behavior of blocks in the buffer cache. The following example illustrates the NOCACHE hint:

```
SELECT /*+ FULL(scott_emp) NOCACHE(scott_emp) */
       ename
FROM scott.emp scott_emp;
```

PUSH_SUBQ The PUSH_SUBQ hint causes nonmerged subqueries to be evaluated at the earliest possible place in the execution plan. Normally, subqueries that are not merged are executed as the last step in the execution plan. If the subquery is relatively inexpensive and reduces the number of rows significantly, it will improve performance to evaluate the subquery earlier.

The hint will have no effect if the subquery is applied to a remote table or one that is joined using a merge join.

Considering
Alternative Syntax

Because SQL is a flexible language, more than one SQL statement may meet the needs of your application. Although two SQL statements may produce the same result, Oracle may process one faster than the other. You can use the results of the EXPLAIN PLAN statement to compare the execution plans and costs of the two statements and determine which is more efficient.

This example shows the execution plans for two SQL statements that perform the same function. Both statements return all the departments in the DEPT table that have no employees in the EMP table. Each statement searches the EMP table with a subquery. Assume there is an index, DEPTNO_INDEX, on the DEPTNO column of the EMP table.

This is the first statement and its execution plan:

```
SELECT dname, deptno
FROM dept
WHERE deptno NOT IN
(SELECT deptno FROM emp);
```

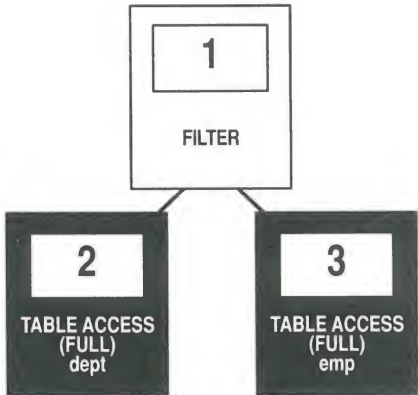


Figure 7 – 3 Execution Plan with Two Full Table Scans

Step 3 of the output indicates that Oracle executes this statement by performing a full table scan of the EMP table despite the index on the DEPTNO column. This full table scan can be a time-consuming operation. Oracle does not use the index because the subquery that searches the EMP table does not have a WHERE clause that makes the index available. However, the following SQL statement selects the same rows by accessing the index:

```
SELECT dname, deptno
FROM dept
WHERE NOT EXISTS
(SELECT deptno
FROM emp
WHERE dept.deptno = emp.deptno);
```

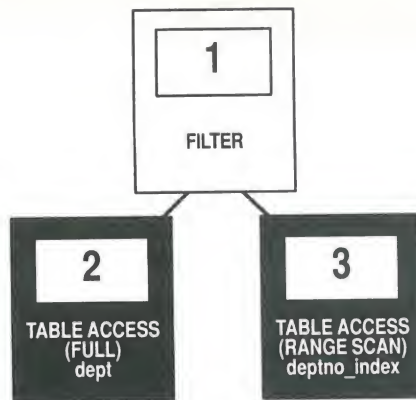



Figure 7 – 4 Execution Plan with a Full Table Scan and an Index Scan

The WHERE clause of the subquery refers to the DEPTNO column of the EMP table, so the index DEPTNO_INDEX is used. The use of the index is reflected in Step 3 of the execution plan. The index range scan of DEPTNO_INDEX takes less time than the full scan of the EMP table in the first statement. Furthermore, the first query performs one full scan of the EMP table for every DEPTNO in the DEPT table. For these reasons, the second SQL statement is faster than the first.

If you have statements in your applications that use the NOT IN operator, as the first query in this example does, you should consider rewriting them so that they use the NOT EXISTS operator. This would allow such statements to use an index, if one exists.

How to Tune Existing SQL Statements

Tuning SQL statements in an existing application is a somewhat, but not altogether, different task from writing new statements. Although much of the required knowledge is the same, the process may be different. You must isolate particular statements in your application to tune. You can do that through these means:

- Familiarize yourself with the application.
- Isolate particular problem statements with the SQL trace facility.

You can then tune these statements individually using the recommendations for writing new SQL statements discussed in the previous section, including using indexes, clusters, hashing, and hints.

Know the Application

You must be familiar with the application, its SQL statements, and its data. If you did not design and develop the application, consult those who did. Find out what the application does:

- what SQL statements the application uses
- what data the application processes
- what are the characteristics and distribution of the data
- what operations the application performs on that data

Discuss performance with application users. Ask them to identify any parts of the application where they feel performance needs to be improved. Narrow these parts down to individual SQL statements, if possible.

Use the SQL Trace Facility

Oracle provides several diagnostic tools for measuring performance. One of the tools especially helpful in tuning applications is the SQL trace facility. The SQL trace facility generates statistics for each SQL statement processed by Oracle. These statistics reflect

- the number of times each SQL statement is parsed, executed, and fetched
- the time necessary to process each SQL statement
- the memory and disk access associated with each SQL statement
- the number of rows each SQL statement processes

The SQL trace facility can also generate execution plans using the EXPLAIN PLAN command.

Run your application with the SQL trace facility enabled. From the resulting statistics, determine which SQL statements take the most time to process. Concentrate your tuning efforts on these statements.

For additional information on how to invoke the SQL trace facility and other performance diagnostic tools and analyze their output, see Appendix A, "Performance Diagnostic Tools".

Tuning Individual SQL Statements

Keep in mind that you can explore alternative syntax for SQL statements without actually modifying your application. Simply use the EXPLAIN PLAN command with the alternative statement that you are considering and compare its execution plan and cost with that of the existing statement. You can find the cost of a SQL statement in the POSITION column of the first row generated by the EXPLAIN PLAN command. However, you must run the application to determine which statement can actually be executed more quickly.

If you create new indexes to tune statements, you can also use the EXPLAIN PLAN command to determine whether the optimizer will choose to use these indexes when the application is run. If you create new indexes to tune a statement that is currently parsed, Oracle invalidates the statement. When the statement is next executed, the optimizer automatically chooses a new execution plan that could potentially use the new index. If you create new indexes on a remote database to tune a distributed statement, the optimizer considers these indexes when the statement is next parsed.

Also keep in mind that the means you use to tune one statement may affect the optimizer's choice of execution plans for others. For example, if you create an index to be used by one statement, the optimizer may choose to use that index for other statements in your application as well. For this reason, you should re-examine your application's performance and rerun the SQL trace facility after you have tuned those statements that you initially identified for tuning.

Bitmap Indexing

This section describes:

- Benefits for Data Warehousing Applications
- What Is a Bitmap Index?
- Bitmap Index Example
- When to Use Bitmap Indexing
- How to Create a Bitmap Index
- Initialization Parameters for Bitmap Indexing
- Bitmap Indexes and EXPLAIN PLAN
- Using Bitmap Access Plans on Regular B-tree Indexes
- Restrictions

Benefits for Data Warehousing Applications

Bitmap indexing benefits data warehousing applications which have large amounts of data and ad hoc queries, but a low level of concurrent transactions. For such applications, bitmap indexing provides:

- reduced response time for large classes of ad hoc queries
- a substantial reduction of space usage compared to other indexing techniques
- dramatic performance gains even on very low end hardware

Fully indexing a large table with a traditional B-tree index can be prohibitively expensive in terms of space since the index can be several times larger than the data in the table. Bitmap indexes are typically only a fraction of the size of the indexed data in the table.

Bitmap indexes are not suitable for OLTP applications with large numbers of concurrent transactions modifying the data. These indexes are primarily intended for decision support in data warehousing applications where users typically query the data rather than update it.

Bitmap indexes are integrated with Oracle's cost-based optimizer and execution engine. They can be used seamlessly in combination with other Oracle execution methods. For example, the optimizer can decide to perform a hash join between two tables using a bitmap index on one table and a regular B-tree index on the other. The optimizer considers bitmap indexes and other available access methods, such as regular B-tree indexes and full table scan, and chooses the most efficient method, taking parallelism into account where appropriate. Note also that parallel query works with bitmap indexes as with traditional indexes. Parallel create index and concatenated indexes are supported.

What Is a Bitmap Index?

Oracle provides four indexing schemes: B-tree indexes (currently the most common), B-tree cluster indexes, hash cluster indexes, and bitmap indexes. These indexing schemes provide complementary performance functionality.

The purpose of an index is to provide pointers to the rows in a table that contain a given key value. For a regular index, this is achieved by storing a list of rowids for each key corresponding to the rows with that key value. (In ORACLE, each key value is stored repeatedly with each stored rowid.) With a bitmap index, a bitmap for each key value is used instead of a list of rowids. Each bit in the bitmap corresponds to a possible rowid, and if the bit is set, it means that the row with the corresponding rowid contains the key value. A mapping function is used to convert the bit position to an actual rowid, so the bitmap index provides the same functionality as a regular index even though it uses a different representation internally. If the number of different key values is small, bitmaps are very space efficient.

Bitmap indexing efficiently merges indexes corresponding to several conditions in the WHERE clause. Rows that satisfy some, but not all the conditions, are filtered out before the table itself is accessed. As a result, response time is improved, often dramatically.

Cardinality

The advantages of using bitmap indexes are greatest for low cardinality columns: that is, columns in which the number of distinct values is small compared to the number of rows in the table. If the values in a column are repeated more than a hundred times, the column is a candidate for a bitmap index. Even columns with a lower number of repetitions (and thus higher cardinality), can be candidates if they tend to be involved in complex conditions in the WHERE clauses of queries.

For example, on a table with one million rows, a column with 10,000 distinct values is a candidate for a bitmap index. A bitmap index on this column can out-perform a B-tree index, particularly when this column is often queried in conjunction with other columns.

B-tree indexes are most effective for high-cardinality data: that is, data with many possible values, such as CUSTOMER_NAME or PHONE_NUMBER. A regular B-tree index can be several times larger than the indexed data. Used appropriately, bitmap indexes can be significantly smaller than a corresponding B-tree index.

In ad hoc queries and similar situations, bitmap indexes can dramatically improve query performance. AND and OR conditions in the WHERE clause of a query can be quickly resolved by performing the corresponding boolean operations directly on the bitmaps before converting the resulting bitmap to rowids. If the resulting number of

rows is small, the query can be answered very quickly without resorting to a full table scan of the table.

Bitmap Index Example The following table shows a portion of a company’s customer data.

CUSTOMER#	MARITAL_STATUS	REGION	GENDER	INCOME_LEVEL
101	single	east	male	bracket_1
102	married	central	female	bracket_4
103	married	west	female	bracket_2
104	divorced	west	male	bracket_4
105	single	central	female	bracket_2
106	married	central	female	bracket_3

Table 7 – 1 Bitmap Index Example

Since MARITAL_STATUS, REGION, GENDER, and INCOME_LEVEL are all low-cardinality columns (there are only three possible values for marital status and region, two possible values for gender, and four for income level) it is appropriate to create bitmap indexes on these columns. A bitmap index should not be created on CUSTOMER# because this is a high-cardinality column. Instead, a unique B-tree index on this column in order would provide the most efficient representation and retrieval.

Table 7 – 2 illustrates the bitmap index for the REGION column in this example. It consists of three separate bitmaps, one for each region.

REGION='east'	REGION='central'	REGION='west'
1	0	0
0	1	0
0	0	1
0	0	1
0	1	0
0	1	0

Table 7 – 2 Sample Bitmap

Each entry (or “bit”) in the bitmap corresponds to a single row of the CUSTOMER table. The value of each bit depends upon the values of the corresponding row in the table. For instance, the bitmap REGION='east' contains a one as its first bit: this is because the region is “east” in the first row of the CUSTOMER table. The bitmap REGION='east' has a zero for its other bits because none of the other rows of the table contain “east” as their value for REGION.

An analyst investigating demographic trends of the company's customers might ask, "How many of our married customers live in the central or west regions?" This corresponds to the following SQL query:

```
SELECT COUNT(*) FROM CUSTOMER WHERE MARITAL_STATUS = 'married' AND
REGION IN ('central','west');
```

Bitmap indexes can process this query with great efficiency by merely counting the number of ones in the resulting bitmap, as illustrated in Figure 7 – 5. To identify the specific customers who satisfy the criteria, the resulting bitmap would be used to access the table.

status = 'married'	region = 'central'	region = 'west'						
0	0	0		0	0		0	
1	1	0		1	1		1	
1	0	1	AND	0	1	OR	1	
0	0	1		0	1		0	
0	1	0		0	1		0	
1	1	0		1	1		1	

Figure 7 – 5 Executing a Query Using Bitmap Indexes

When to Use Bitmap Indexing

Performance Considerations

This section describes three aspects of the indexing scheme you must evaluate when considering whether to use bitmap indexing on a given table: performance, storage, and maintenance.

Bitmap indexes can substantially improve performance of queries with the following characteristics:

- The WHERE clause contains multiple predicates on low- or medium-cardinality columns.
- The individual predicates on these low- or medium-cardinality columns select a large number of rows.
- Bitmap indexes have been created on some or all of these low- or medium-cardinality columns.
- The tables being queried contain many rows.

Multiple bitmap indexes can be used to evaluate the conditions on a single table. Bitmap indexes are thus highly advantageous for complex ad hoc queries that contain lengthy WHERE clauses. Bitmap indexes can also provide optimal performance for aggregate queries.

Storage Considerations

Bitmap indexes can provide considerable storage savings over the use of multiple-column B-tree indexes. In databases that only contain B-tree indexes, a DBA must anticipate the columns that would commonly be accessed together in a single query, and create a multi-column (or concatenated) B-tree index on these columns. Not only would this B-tree index require a large amount of space, but it would also be ordered. That is, a B-tree index on (MARITAL_STATUS, REGION, GENDER) is useless for a query that only accesses REGION and GENDER. To completely index the database, the DBA would have to create indexes on the other permutations of these columns. For the simple case of three low-cardinality columns, there are six possible concatenated B-tree indexes. DBAs must consider the trade-offs between disk space and performance needs when determining which multiple-column B-tree indexes to create.

Bitmap indexes solve this dilemma. Because bitmap indexes can be efficiently combined during query execution, three small single-column bitmap indexes can do the job of six three-column B-tree indexes. Although the bitmap indexes may not be quite as efficient during execution as the appropriate concatenated B-tree indexes, the space savings more than justifies their use.

If a bitmap index is created on a unique key column, it will require more space than a regular B-tree index. However, for columns where each value is repeated hundreds or thousands of times, a bitmap index will typically be less than 25 percent of the size of a regular B-tree index. The bitmaps themselves are stored in compressed format.

Simply comparing the relative sizes of B-tree and bitmap indexes is not an accurate measure, however. Because of their different performance characteristics, you should keep B-tree indexes on high-cardinality data, while creating bitmap indexes on low-cardinality data.

Maintenance Considerations

Bitmap indexes are most appropriate for data warehousing applications. DML and DDL statements such as UPDATE, DELETE, DROP TABLE, and so on, affect bitmap indexes the same way they do traditional indexes: the consistency model is the same. A compressed bitmap for a key value is made up of one or more bitmap segments, each of which is at most half a block in size (but may be smaller). The locking granularity is one such bitmap segment. This may affect performance in environments where many transactions make simultaneous updates.

A B-tree index entry contains a single rowid. Therefore, when the index entry is locked, a single row is locked. With bitmap indexes, an entry can potentially contain a range of rowids. When a bitmap index entry is locked, the entire range of rowids is locked. The number of rowids in this range affects concurrency. For example, a bitmap index on a column

with unique values would lock one rowid per value: concurrency would be the same as for B-tree indexes. As rowids increase in a bitmap segment, concurrency decreases.

Locking issues affect data manipulation language operations, and thus may impact heavy OLTP environments. Locking issues do not, however, affect query performance. As with other types of indexes, updating bitmap indexes is a costly operation. Nonetheless, for bulk inserts and updates where many rows are inserted or many updates are made in a single statement, performance with bitmap indexes can be better than with regular B-tree indexes.

Although bitmap indexes are not appropriate for OLTP applications with a heavy load of concurrent insert, update, and delete operations, their effectiveness in a data warehousing environment is not diminished. In such environments, data is usually maintained via bulk inserts and updates. Index maintenance is deferred until the end of each DML operation. For example, if you insert 1000 rows, the inserted rows are all placed into a sort buffer and then the updates of all 1000 index entries are batched. Thus each bitmap segment is updated only once per DML operation, even if more than one row in that segment changes. This is why `SORT_AREA_SIZE` must be set properly for good performance with inserts and updates on bitmap indexes.

If numerous DML operations have caused increased index size and decreasing performance for queries, you can use the `ALTER INDEX REBUILD` command to compact the index and restore efficient performance.

How to Create a
Bitmap Index

To create a bitmap index, use the BITMAP keyword in the CREATE INDEX command:

CREATE BITMAP INDEX ...

All CREATE INDEX parameters except NOSORT are applicable to bitmap indexes. Multi-column (concatenated) bitmap indexes are supported; they can be defined over at most 14 columns. Other SQL statements concerning indexes, such as DROP, ANALYZE, ALTER, and so on, can refer to bitmap indexes without any extra keyword. The command ANALYZE INDEX VALIDATE STRUCTURE, however, is not applicable to bitmap indexes.

Note: The COMPATIBLE initialization parameter must be set to 7.3.2 or higher, for bitmap indexing to be available.

Uniqueness

System index views USER_INDEXES, ALL_INDEXES, and DBA_INDEXES indicate bitmap indexes by the word BITMAP appearing in the UNIQUENESS column. A bitmap index cannot be declared as UNIQUE and is thus considered a special form of NONUNIQUE index in all cases where uniqueness matters.

Using Hints

The INDEX hint works with bitmap indexes in the same way as with traditional indexes. A new hint is provided, INDEX_COMBINE, which has the same format as the INDEX hint:

INDEX_COMBINE(*table index1 index2 ...*)

If no indexes are given as arguments for this hint, the optimizer will use on the table whatever boolean combination of bitmap indexes has the best cost estimate. If certain indexes are given as arguments, the optimizer will try to use some boolean combination of those particular bitmap indexes.

Performance and Storage
Tips

To obtain optimal performance and disk space usage with bitmap indexes, note the following considerations:

- Large block sizes improve the efficiency of storing, and hence retrieving, bitmap indexes.
- In order to make the compressed bitmaps as small as possible, you should declare NOT NULL constraints on all columns that cannot contain null values.
- Fixed length datatypes are more amenable to a compact bitmap representation than variable length datatypes.

**Initialization
Parameters for Bitmap
Indexing**

The following new initialization parameters have an impact on performance:

CREATE_BITMAP_AREA_SIZE: This parameter determines the amount of memory allocated for bitmap creation. The default value is 8 Mb. A larger value may lead to faster index creation. If cardinality is very small, you can set a small value for this parameter. For example, if cardinality is only 2 then the value can be on the order of kilobytes rather than megabytes. As a general rule, the higher the cardinality, the more memory is needed for optimal performance. This parameter is not dynamically alterable at the session level.

BITMAP_MERGE_AREA_SIZE: This parameter determines the amount of memory used to merge bitmaps retrieved from a range scan of the index. The default value is 1 Mb. A larger value should improve performance because the bitmap segments must be sorted before being merged into a single bitmap. This parameter is not dynamically alterable at the session level.

V733_PLANS_ENABLED determines whether bitmap access paths will be considered for regular indexes on the tables that have at least one bitmap index.

Bitmap Indexes and EXPLAIN PLAN

New index row sources appear in the EXPLAIN PLAN output with the word BITMAP indicating the type. Consider the following sample query and plan, in which the TO ROWIDS option is used to generate the rowids that are necessary for table access.

```
EXPLAIN PLAN FOR
SELECT * FROM T
WHERE
C1 = 2 AND C2 <> 6
OR
C3 BETWEEN 10 AND 20;
SELECT STATEMENT
  TABLE ACCESS              T              BY ROWID
    BITMAP CONVERSION        TO ROWIDS
      BITMAP OR
        BITMAP MINUS
          BITMAP INDEX        C1_IND        SINGLE VALUE
          BITMAP INDEX        C2_IND        SINGLE VALUE
          BITMAP INDEX        C2_IND        SINGLE VALUE
        BITMAP MERGE
          BITMAP INDEX        C3_IND        RANGE SCAN
```

Here, the following new row sources are used:

Row Source	Notes
BITMAP CONVERSION	TO ROWIDS converts the bitmap representation to actual rowids that can be used to access the table. FROM ROWIDS converts the rowids to a bitmap representation. COUNT returns the number of rowids if the actual values are not needed.
BITMAP OR	computes the bitwise OR of two bitmaps
BITMAP MINUS	subtracts the bits of one bitmap from another. This row source is used for negated predicates and can only be used if there are some non-negated predicates yielding a bitmap from which the subtraction can take place. In the example above, such a bitmap results from the predicate c1 = 2. From this bitmap, the bits in the bitmap for c2 = 6 are subtracted. Also, the bits in the bitmap for c2 IS NULL are subtracted, explaining why there are two MINUS row sources in the plan. The NULL subtraction is necessary for semantic correctness unless the column has a NOT NULL constraint.
BITMAP INDEX	SINGLE VALUE looks up the bitmap for a single key value in the index. RANGE SCAN retrieves bitmaps for a key value range. FULL SCAN: A bitmap index full scan is performed if there is no start or stop key.
BITMAP MERGE	merges several bitmaps resulting from a range scan into one bitmap

Table 7 – 3 Bitmap Index Row Sources

Using Bitmap Access
Plans on Regular
B-tree Indexes

If the initialization parameter V733_PLANS_ENABLED is set to TRUE, and there exists at least one bitmap index on the table, the optimizer will consider using a bitmap access path using regular B-tree indexes for that table. This access path may involve combinations of B-tree and bitmap indexes, but might not involve any bitmap indexes at all. However, the optimizer will not generate a bitmap access path using a single B-tree index unless instructed to do so by a hint.

In order to use bitmap access paths for B-tree indexes, the rowids stored in the indexes must be converted to bitmaps. Once such a conversion has taken place, the various boolean operations available for bitmaps can be used. As an example, consider the following query, where there is a bitmap index on column C1, and regular B-tree indexes on columns C2 and C3.

```
EXPLAIN PLAN FOR
SELECT COUNT(*) FROM T
WHERE
C1 = 2 AND C2 = 6
OR
C3 BETWEEN 10 AND 20;
SELECT STATEMENT
  SORT
    BITMAP CONVERSION COUNT
      BITMAP OR
        BITMAP AND
          BITMAP INDEX      C1_IND      SINGLE VALUE
          BITMAP CONVERSION  FROM ROWIDS
            INDEX            C2_IND      RANGE SCAN
          BITMAP CONVERSION  FROM ROWIDS
            SORT              ORDER BY
              INDEX            C3_IND      RANGE SCAN
```

Here, a COUNT option for the BITMAP CONVERSION row source is used to count the number of rows matching the query. There are also conversions FROM ROWIDS in the plan in order to generate bitmaps from the rowids retrieved from the B-tree indexes. The occurrence of the ORDER BY SORT in the plan is due to the fact that the conditions on columns C3 result in more than one list of rowids being returned from the B-tree index. These lists are sorted before they can be converted into a bitmap.

Restrictions

Bitmap indexes have the following restrictions:

For bitmap indexes with direct load, the UNRECOVERABLE and SORTED_INDEX flags are meaningless.

Performing an ALTER TABLE command that adds or modifies a bitmap-indexed column may cause indexes to be invalidated.

The command ANALYZE INDEX VALIDATE STRUCTURE is not applicable to bitmap indexes.

Bitmap indexes are not supported for Trusted Oracle.

Bitmap indexes are not considered by the rule-based optimizer.

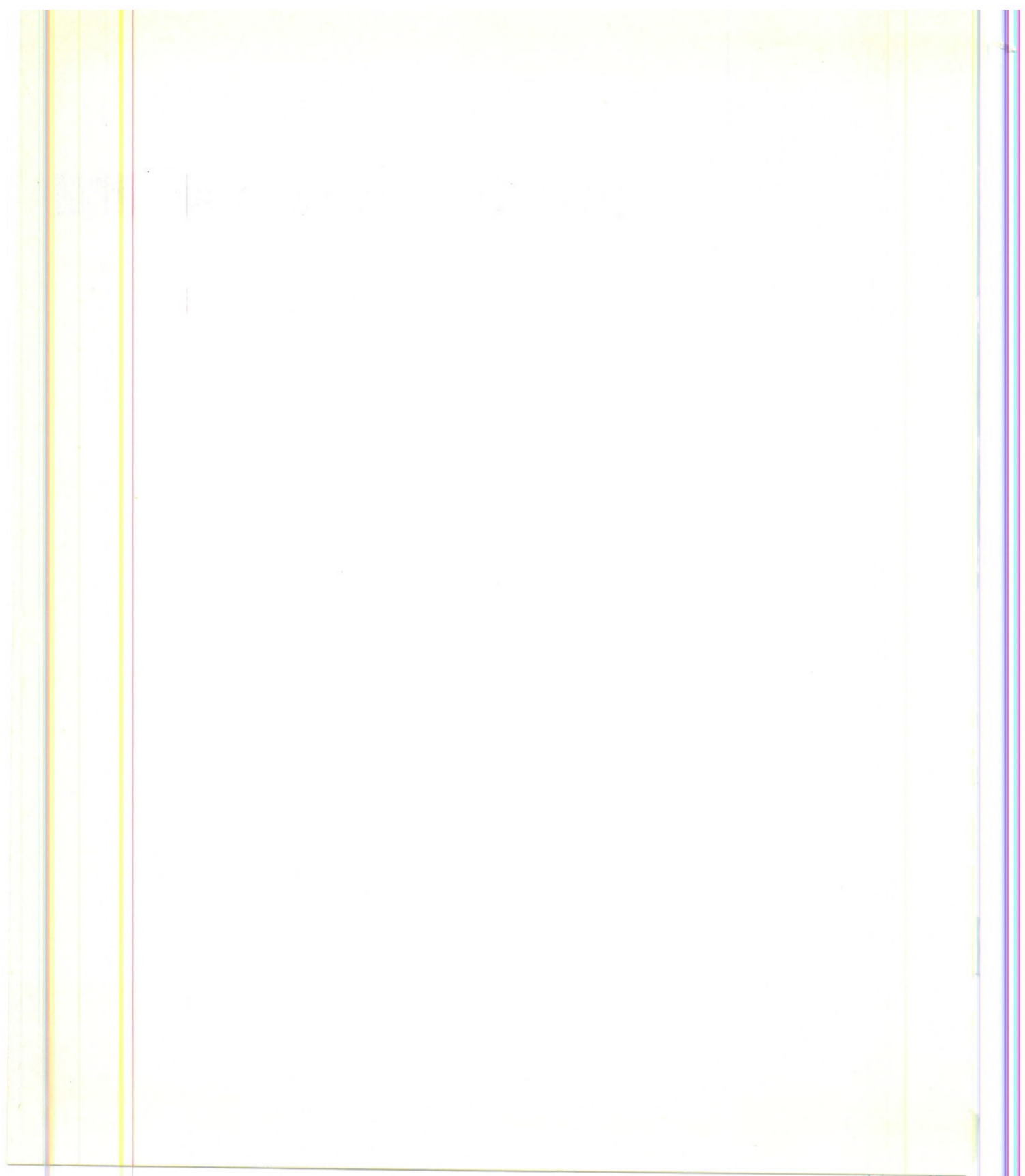
Bitmap indexes cannot be used for referential integrity checking.

For normal indexes, a subquery can be used as an index driver if there is a predicate of the form col = (subquery). Such subqueries cannot be used as keys for a bitmap index.

PART

III

Tuning the Instance



Tuning Memory Allocation

This chapter discusses tuning memory allocation, specifically the following topics:

- the importance of tuning memory
- how to tune memory in general
- tuning your operating system memory
- tuning private SQL and PL/SQL areas
- the shared pool
- the buffer cache
- reallocating memory

Proper sizing of these structures can greatly improve database performance. In this chapter, you will learn how to allocate memory to these structures to achieve best performance.

The Importance of Memory Allocation

Oracle stores information in two places:

- in memory
- on disk

Since memory access is much faster than disk access, it is desirable for data requests to be satisfied by access to memory rather than access to disk. For best performance, it is advantageous to store as much data as possible in memory rather than on disk. However, memory resources on your operating system are likely to be limited. Tuning memory allocation involves distributing available memory to Oracle memory structures.

Because Oracle's memory requirements vary depending on your application, you should tune memory allocation after tuning your application and your SQL statements based on the recommendations presented in Chapter 7, "Tuning SQL Statements". Allocating memory before tuning your application and your SQL statements may make it necessary to resize some Oracle memory structures to meet the needs of your modified statements and application.

Also, you should tune memory allocation before considering the information in Chapter 9, "Tuning I/O". Allocating memory establishes the amount of I/O necessary for Oracle to operate. This chapter shows you how to allocate memory to perform as little I/O as possible. Chapter 9 shows you how to perform that I/O as efficiently as possible.

Steps for Tuning Memory Allocation

This section outlines the process of tuning memory allocation. For best results, you should follow these steps in the order they are presented. Each step is described in more detail in the remainder of this chapter.

Tuning Your Operating System

Ensuring that your operating system runs smoothly and efficiently establishes a solid basis for allocating memory to Oracle. This step also gives you a good idea of the amount of memory on your operating system that is available for Oracle.

Tuning Private SQL and PL/SQL Areas

The use of SQL and PL/SQL areas and the frequency of parse calls are primarily determined by your application. Since parsing affects the frequency of access to the data dictionary, you should tune private SQL and PL/SQL areas before tuning the data dictionary cache in the shared pool.

Tuning the Shared Pool Tuning the shared pool involves allocating memory for these memory structures:

- the library cache containing shared SQL and PL/SQL areas
- the data dictionary cache
- information for sessions connected through shared server processes

A cache miss on the data dictionary cache or library cache is more expensive than a miss on the buffer cache. For this reason, you should allocate sufficient memory for the shared pool first.

Tuning the Buffer Cache

After tuning private SQL and PL/SQL areas and the shared pool, you can devote the remaining available memory to the buffer cache.

It may be necessary to repeat the steps of memory allocation after the initial pass through the process. Subsequent passes allow you to make adjustments in earlier steps based on changes in later steps. For example, if you increase the size of the buffer cache, you may need to allocate more memory to Oracle to avoid paging and swapping.

Tuning Your Operating System

You should begin tuning memory allocation by tuning your operating system with these goals:

- to reduce paging and swapping
- to fit the System Global Area (SGA) into main memory
- to allocate enough memory to individual users

These goals apply in general to most operating systems. However, the details of tuning your operating system vary depending on which operating system you are using.



OSDoc

Additional Information: Refer to your operating system hardware and software documentation as well as your Oracle operating system-specific documentation for more information on tuning your operating system memory usage.

Reducing Paging and Swapping

Your operating system may store information in any of these places:

- real memory
- virtual memory
- expanded storage
- disk

Your operating system may also move information from one storage location to another. Depending on your operating system, this movement is called paging or swapping. Many operating systems page and swap to accommodate large amounts of information that do not fit into real memory. However, paging and swapping take time. Excessive paging or swapping can reduce the performance of many operating systems.

Monitor your operating system behavior with operating system utilities. Excessive paging or swapping indicates that new information is often being moved into memory. In this case, your system's total memory may not be large enough to hold everything for which you have allocated memory. You should either increase the total memory on your system or decrease the amount of memory you have allocated.

Tuning the System Global Area (SGA)

Since the purpose of the System Global Area (SGA) is to store data in memory for fast access, the SGA should always be contained in main memory. If the SGA is swapped to disk, its data is no longer so quickly accessible. On most operating systems, the disadvantage of excessive paging significantly outweighs the advantage of a large SGA, so you should ensure that the entire SGA always fits into memory and is not paged or swapped.

You can cause Oracle to read the entire SGA into memory when you start your instance by setting the value of the initialization parameter `PRE_PAGE_SGA` to `YES`. This setting may increase the amount of time necessary for instance startup, but it is likely to decrease the amount of time necessary for Oracle to reach its full performance capacity after startup. Note that this setting does not prevent your operating system from paging or swapping the SGA after it is initially read into memory.

You can see how much memory is allocated to the SGA and each of its internal structures by issuing this Server Manager statement:

```
SVRMGR> SHOW SGA
```

The output of this statement might look like this:

Total System Global Area	3554188 bytes
Fixed Size	22208 bytes
Variable Size	3376332 bytes
Database Buffers	122880 bytes
Redo Buffers	32768 bytes

Information about the SGA can also be obtained through SNMP.

Some operating systems for IBM mainframe computers are equipped with *expanded storage*, or special memory, in addition to main memory to which paging can be performed very quickly. These operating systems may be able to page data between main memory and expanded storage faster than Oracle can read and write data between the SGA and disk. For this reason, allowing a larger SGA to be swapped may lead to better performance than ensuring that a smaller SGA stays in main memory. If your operating system has expanded storage, you can take advantage of it by allocating a larger SGA despite the resulting paging.

User Memory Allocation

On some operating systems, you may have control over the amount of physical memory allocated to each user. Be sure all users are allocated enough memory to accommodate the resources they need to use their application with Oracle. Depending on your operating system, these resources may include

- the Oracle executable image
- the SGA
- Oracle application tools
- application-specific data

On some operating systems, Oracle software can be installed so that a single executable image can be shared by many users. By sharing executable images among users, you can reduce the amount of memory required by each user.

Tuning Private SQL and PL/SQL Areas

In this section you learn how to tune private SQL and PL/SQL areas. Tuning private SQL areas involves identifying unnecessary parse calls made by your application and then reducing them. To reduce parse calls, you may have to increase the number of private SQL areas that your application can have allocated at once. Throughout this section, information about private SQL areas and SQL statements also applies to private PL/SQL areas and PL/SQL blocks.

Identifying Unnecessary Parse Calls

To identify unnecessary parse calls, run your application with the SQL trace facility enabled. For each SQL statement in the trace output, examine the *count* statistic for the Parse step. This statistic tells you how many times your application makes a parse call for the statement. This statistic includes parse calls that are satisfied by access to the library cache as well as parse calls that result in actually parsing the statement.

Note: This statistic does not include implicit parsing that occurs when an application executes a statement whose shared SQL area is no longer in the library cache. For information on detecting implicit parsing, see the section “Examining Library Cache Activity” on page 8 – 9.

If the *count* value for the Parse step is near the *count* value for the Execute step for a statement, your application may be deliberately making a parse call each time it executes the statement. Try to reduce these parse calls through your application tool.

Reducing Unnecessary Parse Calls

Depending on the Oracle application tool you are using, you may be able to control how frequently your application performs parse calls and allocates and deallocates private SQL areas. Whether your application reuses private SQL areas for multiple SQL statements determines how many parse calls your application performs and how many private SQL areas the application requires.

In general, an application that reuses private SQL areas for multiple SQL statements does not need as many private SQL areas as an application that does not reuse private SQL areas. However, an application that reuses private SQL areas must perform more parse calls because the application must make a new parse call whenever an existing private SQL is reused for a new SQL statement.

Be sure that your application can open enough private SQL areas to accommodate all of your SQL statements. If you allocate more private SQL areas, you may need to increase the limit on the number of cursors permitted for a session. You can increase this limit by increasing the value of the initialization parameter OPEN_CURSORS. The maximum value for this parameter depends on your operating system. The minimum value is 5.

The means by which you control parse calls and allocation and deallocation of private SQL areas varies depending on your Oracle application tool. The following sections introduce the means used for some tools. Note that these means apply only to private SQL areas and not to shared SQL areas.

Reducing Parse Calls with the Oracle Precompilers With the Oracle Precompilers, you control private SQL areas and parse calls with these options:

- HOLD_CURSOR
- RELEASE_CURSOR
- MAXOPENCURSORS

These options can be specified in two ways:

- on the precompiler command line
- within the precompiler program

With these options, you can employ different strategies for managing private SQL areas during the course of the program.

For information on these options, see the *Programmer's Guide to the Oracle Precompilers*.

Reducing Parse Calls with the Oracle Call Interfaces (OCIs) With the Oracle Call Interface (OCI), you have complete control over parse calls and private SQL areas with these OCI calls:

OSQL3	An OSQL3 or OPARSE call allocates a private SQL area for a SQL statement.
OPARSE	
OCLOSE	An OCLOSE call closes a cursor and deallocates the private SQL area of its associated statement.

For more information on these calls, see the *Programmer's Guide to the Oracle Call Interface*.

Reducing Parse Calls with Oracle Forms With Oracle Forms, you also have some control over whether your application reuses private SQL areas. You can exercise this control in three places:

- at the trigger level
- at the form level
- at runtime

For more information on the reuse of private SQL areas by Oracle Forms, see the *Oracle Forms Reference* manual.

Tuning the Shared Pool

This section describes tuning these parts of the shared pool:

- library cache
- data dictionary cache
- session information

This section presents these parts of the shared pool in order of importance. Because the algorithm that Oracle uses to manage data in the shared pool tends to hold dictionary data in memory longer than library cache data, tuning the library cache to an acceptable cache hit ratio often ensures that the data dictionary cache hit ratio is also acceptable. Allocating space in the shared pool for session information is only necessary if you are using the multi-threaded server architecture.

Tuning the Library Cache

The library cache contains shared SQL and PL/SQL areas. This section tells you how to tune the library cache by

- examining library cache activity
- reducing library cache misses
- speeding access to shared SQL and PL/SQL areas in the library cache

Throughout this section, information about shared SQL areas and SQL statements also applies to shared PL/SQL areas and PL/SQL blocks.

Library cache misses can occur on either of these steps in the processing of a SQL statement.

Parse If an application makes a parse call for a SQL statement and the parsed representation of the statement does not already exist in a shared SQL area in the library cache, Oracle parses the statement and allocates a shared SQL area. You may be able to reduce library cache misses on parse calls by ensuring that SQL statements can share a shared SQL area whenever possible.

Execute If an application makes an execute call for a SQL statement and the shared SQL area containing the parsed representation of the statement has been deallocated from the library cache to make room for another statement, Oracle implicitly reparses the statement, allocates a new shared SQL area for it, and executes it. You may be able to reduce library cache misses on execution calls by allocating more memory to the library cache.

Determine whether misses on the library cache are affecting the performance of Oracle by querying the dynamic performance table V\$LIBRARYCACHE.

The V\$LIBRARYCACHE Table Statistics reflecting library cache activity are kept in the dynamic performance table V\$LIBRARYCACHE. These statistics reflect all library cache activity since the most recent instance startup. To monitor library cache activity, examine this table. By default, this table is only available to the user SYS and to users granted SELECT ANY TABLE system privilege, such as SYSTEM.

Each row in this table contains statistics for one type of item kept in the library cache. The item described by each row is identified by the value of the NAMESPACE column. Rows of the table with these NAMESPACE values reflect library cache activity for SQL statements and PL/SQL blocks:

- 'SQL AREA'
- 'TABLE/PROCEDURE'
- 'BODY'
- 'TRIGGER'

Rows with other NAMESPACE values reflect library cache activity for object definitions that Oracle uses for dependency maintenance.

These columns of the V\$LIBRARYCACHE table reflect library cache misses on execution calls:

PINS	This column shows the number of times an item in the library cache was executed.
RELOADS	This column shows the number of library cache misses on execution steps.

Information in the V\$LIBRARYCACHE table can also be obtained through SNMP.

Querying the V\$LIBRARYCACHE Table Monitor the statistics in the V\$LIBRARYCACHE table over a period of time with this query:

```
SELECT SUM(pins) "Executions",
       SUM(reloads) "Cache Misses while Executing"
FROM v$librarycache;
```

The output of this query might look like this:

Executions	Cache Misses while Executing
320871	549

Interpreting the V\$LIBRARYCACHE Table Examining the data returned by the sample query leads to these observations:

- The sum of the PINS column indicates that SQL statements, PL/SQL blocks, and object definitions were accessed for execution a total of 320,871 times.
- The sum of the RELOADS column indicates that 549 of those executions resulted in library cache misses causing Oracle to implicitly reparse a statement or block or reload an object definition because it had aged out of the library cache.
- The ratio of the total RELOADS to total PINS is about 0.17%. This value means that only 0.17% of executions resulted in reparsing.

Total RELOADS should be near 0. If the ratio of RELOADS to PINS is more than 1%, then you should reduce these library cache misses through the means discussed in the next section.

Reducing Library Cache Misses

You can reduce library cache misses by

- allocating additional memory for the library cache
- writing identical SQL statements whenever possible

Allocating Additional Memory for the Library Cache You may be able to reduce library cache misses on execution calls by allocating additional memory for the library cache. To ensure that shared SQL areas remain in the cache once their SQL statements are parsed, increase the amount of memory available to the library cache until the V\$LIBRARYCACHE.RELOADS value is near 0. To increase the amount of memory available to the library cache, increase the value of the initialization parameter SHARED_POOL_SIZE. The maximum value for this parameter depends on your operating system. This measure will reduce implicit reparsing of SQL statements and PL/SQL blocks on execution.

To take advantage of additional memory available for shared SQL areas, you may also need to increase the number of cursors permitted for a session. You can increase this limit by increasing the value of the initialization parameter OPEN_CURSORS.

Be careful not to induce paging and swapping by allocating too much memory for the library cache. The benefits of a library cache large enough to avoid cache misses can be partially offset by reading shared SQL areas into memory from disk whenever you need to access them.

Writing Identical SQL Statements You may be able to reduce library cache misses on parse calls by ensuring that SQL statements and PL/SQL blocks share a shared SQL area whenever possible. For two different occurrences of a SQL statement or PL/SQL block to share a shared SQL area, they must be identical according to these criteria:

- The text of the SQL statements or PL/SQL blocks must be identical, character for character, including spaces and case.

For example, these statements cannot use the same shared SQL area:

```
SELECT * FROM emp;  
SELECT *   FROM emp;
```

These statements cannot use the same shared SQL area:

```
SELECT * FROM emp;  
SELECT * FROM Emp;
```

- References to schema objects in the SQL statements or PL/SQL blocks must resolve to the same object in the same schema.

For example, if the schemas of the users BOB and ED both contain an EMP table and both users issue the following statement, their statements cannot use the same shared SQL area:


```
SELECT * FROM emp;  
SELECT * FROM emp;
```

If both statements query the same table and qualify the table with the schema, as in the following statement, then they can use the same shared SQL area:

```
SELECT * FROM bob.emp;
```

- Bind variables in the SQL statements must match in name and datatype. For example, these statements cannot use the same shared SQL area:

```
SELECT * FROM emp WHERE deptno = :department_no;  
SELECT * FROM emp WHERE deptno = :d_no;
```

- The SQL statements must be optimized using the same optimization approach and, in the case of the cost-based approach, the same optimization goal. For information on optimization approach and goal, see Chapter 7, "Tuning SQL Statements".

Shared SQL areas are most useful for reducing library cache misses for multiple users running the same application. Discuss these criteria with the developers of such applications and agree on strategies to ensure that the SQL statements and PL/SQL blocks of an application can use the same shared SQL areas:

- Use bind variables rather than explicitly specified constants in your statements whenever possible.

For example, the following two statements cannot use the same shared area because they do not match character for character:

```
SELECT ename, empno FROM emp WHERE deptno = 10;  
SELECT ename, empno FROM emp WHERE deptno = 20;
```

You can accomplish the goals of these statements by using the following statement that contains a bind variable, binding 10 for one occurrence of the statement and 20 for the other:

```
SELECT ename, empno FROM emp WHERE deptno = :department_no;
```

The two occurrences of the statement can use the same shared SQL area.

- Be sure that individual users of the application do not change the optimization approach and goal for their individual sessions.

You can also increase the likelihood that SQL statements issued by different applications can share SQL areas by establishing these policies among the developers of these applications:

- Standardize naming conventions for bind variables and spacing conventions for SQL statements and PL/SQL blocks.
- Use stored procedures whenever possible. Multiple users issuing the same stored procedure automatically use the same shared PL/SQL area. Since stored procedures are stored in a parsed form, they eliminate runtime parsing altogether.

Speeding Access to Shared SQL Areas on Execution Calls

If you have no library cache misses, you may still be able to speed execution calls by setting the value of the initialization parameter `CURSOR_SPACE_FOR_TIME`. This parameter specifies when a shared SQL area can be deallocated from the library cache to make room for a new SQL statement. The default value of this parameter is `FALSE`, meaning that a shared SQL area can be deallocated from the library cache regardless of whether application cursors associated with its SQL statement are open. The value of `TRUE` means that a shared SQL area can only be deallocated when all application cursors associated with its statement are closed.

Depending on the value of `CURSOR_SPACE_FOR_TIME`, Oracle behaves differently when an application makes an execution call. If the value is `FALSE`, Oracle must take time to check that a shared SQL area containing the SQL statement is in the library cache. If the value is `TRUE`, Oracle need not make this check because the shared SQL area can never be deallocated while an application cursor associated with it is open. Setting the value of the parameter to `TRUE` saves Oracle a small amount of time and may slightly improve the performance of execution calls. This value also prevents the deallocation of private SQL areas until associated application cursors are closed.

Do not set the value of `CURSOR_SPACE_FOR_TIME` to `TRUE` if there are library cache misses on execution calls. Such library cache misses indicate that the shared pool is not large enough to hold the shared SQL areas of all concurrently open cursors. If the value is `TRUE` and there is no space in the shared pool for a new SQL statement, the statement cannot be parsed and Oracle returns an error saying that there is no more shared memory. If the value is `FALSE` and there is no space for a new statement, Oracle deallocates an existing shared SQL area. Although deallocating a shared SQL area results in a library cache miss later, it is preferable to an error halting your application because a SQL statement cannot be parsed.

Do not set the value of `CURSOR_SPACE_FOR_TIME` to `TRUE` if the amount of memory available to each user for private SQL areas is scarce. This value also prevents the deallocation of private SQL areas associated with open cursors. If the private SQL areas for all concurrently open cursors fills the user's available memory so that there is no space to allocate a private SQL area for a new SQL statement, the statement cannot be parsed and Oracle returns an error indicating that there is not enough memory.

Caching Session Cursors

If an application repeatedly issues parse calls on the same set of SQL statements, the reopening of the session cursors can affect system performance. Session cursors can be stored in a session cursor cache. This feature can be particularly useful for applications designed using Oracle Forms because switching between forms closes all session cursors associated with a form.

Oracle uses the shared SQL area to determine if more than three parse requests have been issued on a given statement. If so, Oracle assumes the session cursor associated with the statement should be cached and moves the cursor into the session cursor cache. Subsequent requests to parse that SQL statement by the same session will then find the cursor in the session cursor cache.

To enable caching of session cursors, you must set the initialization parameter `SESSION_CACHED_CURSORS`. This parameter is a positive integer that specifies the maximum number of session cursors kept in the cache. A least recently used (LRU) algorithm ages out entries in the session cursor cache to make room for new entries when needed.

You can also enable the session cursor cache dynamically with the `ALTER SESSION SET SESSION_CACHED_CURSORS` command.

To determine whether the session cursor cache is sufficiently large for your instance, you can examine the session statistic "session cursor cache hits" in the `V$SESSTAT` view. This statistic counts the number of times a parse call found a cursor in the session cursor cache. If this statistic is a relatively low percentage of the total parse call count for the session, you should consider setting `SESSION_CACHED_CURSORS` to a larger value.

Tuning the Data Dictionary Cache

In this section, you will tune the data dictionary cache. These topics are discussed in this section:

- how to monitor the activity of the data dictionary cache
- how to improve the performance of the data dictionary cache

Determine whether misses on the data dictionary cache are affecting the performance of Oracle. You can examine cache activity by querying the V\$ROWCACHE table as described in the following sections.

Misses on the data dictionary cache are to be expected in some cases. Upon instance startup, the data dictionary cache contains no data, so any SQL statement issued is likely to result in cache misses. As more data is read into the cache, the likelihood of cache misses should decrease. Eventually the database should reach a “steady state” in which the most frequently used dictionary data is in the cache. At this point, very few cache misses should occur. To tune the cache, examine its activity only after your application has been running.

The V\$ROWCACHE View Statistics reflecting data dictionary activity are kept in the dynamic performance table V\$ROWCACHE. By default, this table is only available to the user SYS and to users granted SELECT ANY TABLE system privilege, such as SYSTEM.

Each row in this table contains statistics for a single type of the data dictionary item. These statistics reflect all data dictionary activity since the most recent instance startup. These columns in the V\$ROWCACHE table reflect the use and effectiveness of the data dictionary cache:

PARAMETER	<p>This column identifies a particular data dictionary item. For each row, the value in this column is the item prefixed by 'dc_'.</p> <p>For example, in the row that contains statistics for file descriptions, this column has the value 'dc_files'.</p>
GETS	<p>This column shows the total number of requests for information on the corresponding item.</p> <p>For example, in the row that contains statistics for file descriptions, this column has the total number of requests for file descriptions data.</p>
GETMISSES	<p>This column shows the number of data requests resulting in cache misses.</p>

Querying the V\$ROWCACHE Table Monitor the statistics in the V\$ROWCACHE table over a period of time while your application is running with this query:

```
SELECT SUM(gets) "Data Dictionary Gets",  
       SUM(getmisses) "Data Dictionary Cache Get Misses"  
FROM v$rowcache;
```

The output of this query might look like this:

Data Dictionary Gets	Data Dictionary Cache Get Misses
1439044	3120

Interpreting the V\$ROWCACHE Table Examining the data returned by the sample query leads to these observations:

- The sum of the GETS column indicates that there were a total of 1,439,044 requests for dictionary data.
- The sum of the GETMISSES column indicates that 3120 of the requests for dictionary data resulted in cache misses.
- The ratio of the sums of GETMISSES to GETS is about 0.2%.

Reducing Data Dictionary Cache Misses

Examine cache activity by monitoring the sums of the GETS and GETMISSES columns. For frequently accessed dictionary caches, the ratio of total GETMISSES to total GETS should be less than 10% or 15%. If this ratio continues to increase above this threshold while your application is running, you should consider increasing the amount of memory available to the data dictionary cache. To increase the memory available to the cache, increase the value of the initialization parameter SHARED_POOL_SIZE. The maximum value for this parameter varies depending on your operating system.

Tuning the Shared Pool with the Multi-Threaded Server

In the multi-threaded server architecture, Oracle stores session information in the shared pool rather than in the memory of user processes. Session information includes private SQL areas and sort areas. If you are using the multi-threaded server architecture, you may need to make your shared pool larger to accommodate session information. You can increase the size of the shared pool by increasing the value of the SHARED_POOL_SIZE initialization parameter. This section discusses measuring the size of session information by querying the dynamic performance table V\$SESSTAT.

The V\$SESSTAT Table

Oracle collects statistics on total memory used by a session and stores them in the dynamic performance table V\$SESSTAT. By default, this table is only available to the user SYS and to users granted SELECT ANY TABLE system privilege, such as SYSTEM. These statistics are useful for measuring session memory use:

<i>session memory</i>	The value of this statistic is the amount of memory in bytes allocated to the session.
<i>max session memory</i>	The value of this statistic is the maximum amount of memory in bytes ever allocated to the session.

Querying the V\$SESSTAT Table

You can use this query to decide how much larger to make the shared pool if you are using the multi-threaded server. Issue these queries while your application is running:

```
SELECT SUM(value) || ' bytes' "Total memory for all sessions"
FROM v$sesstat, v$statname
WHERE name = 'session memory'
AND v$sesstat.statistic# = v$statname.statistic#;
```

```
SELECT SUM(value) || ' bytes' "Total max mem for all sessions"
FROM v$sesstat, v$statname
WHERE name = 'max session memory'
AND v$sesstat.statistic# = v$statname.statistic#;
```

These queries also select from the dynamic performance table V\$STATNAME to obtain internal identifiers for *session memory* and *max session memory*. The results of these queries might look like this:

Total memory for all sessions
157125 bytes
Total max mem for all sessions
417381 bytes

Interpreting the V\$SESSTAT Table

The result of the first query indicates that the memory currently allocated to all sessions is 157,125 bytes. This value is the total memory whose location depends on how the sessions are connected to Oracle. If the sessions are connected to dedicated servers processes, this memory is part of the memories of the user processes. If the sessions are connected to shared server processes, this memory is part of the shared pool. The result of the second query indicates the sum of the maximum sizes of the memories for all sessions is 417,381 bytes. The second result is greater than the first because some sessions have deallocated memory since allocating their maximum amounts.

You can use the result of either of these queries to determine how much larger to make the shared pool if you use the multi-threaded server. The first value is likely to be a better estimate than the second unless nearly all sessions are likely to reach their maximum allocations at the same time.

Tuning the Buffer Cache

In this section, you will learn how to tune the buffer cache. The following issues are covered in this section:

- how to monitor buffer cache performance
- how to improve buffer cache performance

Examining Buffer Cache Activity

Oracle collects statistics that reflect data access and stores them in the dynamic performance table V\$SYSSTAT. By default, this table is only available to the user SYS and to users granted SELECT ANY TABLE system privilege, such as SYSTEM. These statistics are useful for tuning the buffer cache:

<i>db block gets, consistent gets</i>	The sum of the values of these statistics is the total number of requests for data. This value includes requests satisfied by access to buffers in memory.
<i>physical reads</i>	The value of this statistic is the total number of requests for data resulting in access to datafiles on disk.

Monitor these statistics over a period of time while your application is running with this query:

```
SELECT name, value
FROM v$sysstat
WHERE name IN ('db block gets', 'consistent gets',
              'physical reads');
```

The output of this query might look like this:

NAME	VALUE
db block gets	85792
consistent gets	278888
physical reads	23182

Calculate the hit ratio for the buffer cache with this formula:

Hit Ratio = 1 – (*physical reads* / (*db block gets* + *consistent gets*))

Based on the statistics obtained by the example query, the buffer cache hit ratio is 94%.

Information in the V\$SYSSTAT table can also be obtained through SNMP.

Reducing Buffer Cache Misses

If your hit ratio is low, say less than 60% or 70%, then you may want to increase the number of buffers in the cache to improve performance. To make the buffer cache larger, increase the value of the initialization parameter `DB_BLOCK_BUFFERS`.

Oracle can collect statistics that estimate the performance gain that would result from increasing the size of your buffer cache. With these statistics, you can estimate how many buffers to add to your cache.

The X\$KCBRBH Table

The virtual table `SYS.X$KCBRBH` contains statistics that estimate the performance of a larger cache. Each row in the table reflects the relative performance value of adding a buffer to the cache. This table can only be accessed by the user `SYS`. The following are the columns of the `X$KCBRBH` table:

INDX	The value of this column is one less than the number of buffers that would potentially be added to the cache.
COUNT	The value of this column is the number of additional cache hits that would be obtained by adding buffer number <code>INDX+1</code> to the cache.

For example, in the first row of the table, the `INDX` value is 0 and the `COUNT` value is the number of cache hits to be gained by adding the first additional buffer to the cache. In the second row, the `INDX` value is 1 and the `COUNT` value is the number of cache hits for the second additional buffer.

Enabling the X\$KCBRBH Table

The collection of statistics in the `X$KCBRBH` table is controlled by the initialization parameter `DB_BLOCK_LRU_EXTENDED_STATISTICS`. The value of this parameter determines the number of rows in the `X$KCBRBH` table. The default value of this parameter is 0, which means the default behavior is to not collect statistics.

To enable the collection of statistics in the `X$KCBRBH` table, set the value of `DB_BLOCK_LRU_EXTENDED_STATISTICS`. For example, if you set the value of the parameter to 100, Oracle will collect 100 rows of statistics, each row reflecting the addition of one buffer, up to 100 extra buffers.

Collecting these statistics incurs some performance overhead. This overhead is proportional to the number of rows in the table. To avoid this overhead, collect statistics only when you are tuning the buffer cache and disable the collection of statistics when you are finished tuning.

Querying the X\$KCBRBH Table

From the information in the X\$KCBRBH table, you can predict the potential gains of increasing the cache size. For example, to determine how many more cache hits would occur if you added 20 buffers to the cache, query the X\$KCBRBH table with the following SQL statement:

```
SELECT SUM(count) ach
FROM sys.x$kcbrbh
WHERE indx < 20;
```

You can also determine how these additional cache hits would affect the hit ratio. Use the following formula to calculate the hit ratio based on the values of the statistics *db block gets*, *consistent gets*, and *physical reads* and the number of additional cache hits (ACH) returned by the query:

$$\text{Hit Ratio} = 1 - (\text{physical reads} - \text{ACH} / (\text{db block gets} + \text{consistent gets}))$$

Grouping Rows in the X\$KCBRBH Table

Another way to examine the X\$KCBRBH table is to group the additional buffers in large intervals. You can query the table with a SQL statement similar to this one:

```
SELECT 250*TRUNC(indx/250)+1||' to '||250*(TRUNC(indx/250)+1)
"Interval", SUM(count) "Buffer Cache Hits"
FROM sys.x$kcbrbh
GROUP BY TRUNC(indx/250);
```

The result of this query might look like

Interval	Buffer Cache Hits
1 to 250	16080
251 to 500	10950
501 to 750	710
751 to 1000	23140

where:

INTERVAL	Is the interval of additional buffers to be added to the cache.
BUFFER CACHE HITS	Is the number of additional cache hits to be gained by adding the buffers in the INTERVAL column.

Examining the query output leads to these observations:

- If 250 buffers were added to the cache, 16,080 cache hits would be gained.
- If 250 more buffers were added for a total of 500 additional buffers, 10,950 cache hits would be gained in addition to the 16,080 cache hits from the first 250 buffers. This means that adding 500 buffers would yield a total of 27,030 additional cache hits.
- If 250 more buffers were added for a total of 750 additional buffers, 710 cache hits would be gained, yielding a total of 27,740 additional cache hits.
- If 250 buffers were added to the cache for a total of 1000 additional buffers, 23,140 cache hits would be gained, yielding a total of 50,880 additional cache hits.

Based on these observations, you should decide how many buffers to add to the cache. In this case, you may make these decisions:

- It is wise to add 250 or 500 buffers, provided memory resources are available. Both of these increments offer significant performance gains.
- It is unwise to add 750 buffers. Nearly the entire performance gain made by such an increase can be made by adding 500 buffers instead. Also, the memory allocated to the additional 250 buffers may be better used by some other Oracle memory structure.
- It is wise to add 1000 buffers, provided memory resources are available. The performance gain from adding 1000 buffers to the cache is significantly greater than the gains from adding 250, 500, or 750 buffers.

Removing Unnecessary Buffers

If your hit ratio is high, your cache is probably large enough to hold your most frequently accessed data. In this case, you may be able to reduce the cache size and still maintain good performance. To make the buffer cache smaller, reduce the value of the initialization parameter `DB_BLOCK_BUFFERS`. The minimum value for this parameter is 4. You can apply any leftover memory to other Oracle memory structures.

Oracle can collect statistics to predict buffer cache performance based on a smaller cache size. Examining these statistics can help you determine how small you can afford to make your buffer cache without adversely affecting performance.

The X\$KCBCBH Table

The virtual table SYS.X\$KCBCBH contains the statistics that estimate the performance of a smaller cache. The X\$KCBCBH table is similar in structure to the X\$KCBRBH table. This table can only be accessed by the user SYS. The following are the columns of the X\$KCBCBH table:

INDX	The value of this column is the potential number of buffers in the cache.
COUNT	The value of this column is the number of cache hits attributable to buffer number INDX.

The number of rows in this table is equal to the number of buffers in your buffer cache. Each row in the table reflects the number of cache attributed to a single buffer. For example, in the second row, the INDX value is 1 and the COUNT value is the number of cache hits for the second buffer. In the third row, the INDX value is 2 and the COUNT value is the number of cache hits for the third buffer.

The first row of the table contains special information. The INDX value is 0 and the COUNT value is the total number of blocks moved into the first buffer in the cache.

Enabling the X\$KCBCBH Table

The collection of statistics in the X\$KCBCBH table is controlled by the initialization parameter DB_BLOCK_LRU_STATISTICS. The value of this parameter determines whether Oracle collects the statistics. The default value for this parameter is FALSE, which means that the default behavior is not to collect statistics.

- To enable the collection of statistics in the X\$KCBCBH table, set the value of DB_BLOCK_LRU_STATISTICS to TRUE.
- Collecting these statistics incurs some performance overhead. To avoid this overhead, collect statistics only when you are tuning the buffer cache and disable the collection of statistics when you are finished tuning.

Querying the X\$KCBCBH Table

From the information in the X\$KCBCBH table, you can predict the number of additional cache misses that would occur if the number of buffers in the cache were reduced. If your buffer cache currently contains 100 buffers, you may want to know how many more cache misses would occur if it had only 90. To determine the number of additional cache misses, query the X\$KCBCBH table with the SQL statement:

```
SELECT SUM(count) acm
FROM sys.x$kcbbh
WHERE indx >= 90;
```

You can also determine the hit ratio based on this cache size. Use the following formula to calculate the hit ratio based on the values of the statistics *db block gets*, *consistent gets*, and *physical reads* and the number of additional cache misses (ACM) returned by the query:

$$\text{Hit Ratio} = 1 - (\text{physical reads} + \text{ACM} / (\text{db block gets} + \text{consistent gets}))$$

Another way to examine the X\$KCB CBH table is to group the buffers in intervals. For example, if your cache contains 100 buffers, you may want to divide the cache into four 25-buffer intervals. You can query the table with a SQL statement similar to this one:

```
SELECT 25*TRUNC(indx/25)+1||' to '||25*(TRUNC(indx/25)+1)
"Interval", SUM(count) "Buffer Cache Hits"
FROM sys.x$kcbbch
WHERE indx > 0 GROUP BY TRUNC(indx/25);
```

Note that the WHERE clause prevents the query from collecting statistics from the first row of the table. The result of this query might look like

Interval	Buffer Cache Hits
1 to 25	1900
26 to 50	1100
51 to 75	1360
76 to 100	230

where:

INTERVAL	Is the interval of buffers in the cache.
BUFFER CACHE HITS	Is the number of cache hits attributable to the buffers in the INTERVAL column.

Examining the query output leads to these observations:

- The last 25 buffers in the cache (buffers 76 to 100) contribute 230 cache hits. If the cache were reduced in size by 25 buffers, 230 cache hits would be lost.
- The third 25-buffer interval (buffers 51 to 75) contributes 1,360 cache hits. If these buffers were removed from the cache, 1,360 cache hits would be lost in addition to the 230 cache hits lost for buffers 76 to 100. Removing 50 buffers would result in losing a total of 1,590 cache hits.
- The second 25-buffer interval (buffers 26 to 50) contributes 1,100 cache hits. Removing 75 buffers from the cache would result in losing a total of 2,690 cache hits.

- The first 25 buffers in the cache (buffers 1 to 25) contribute 1,900 cache hits.

Based on these observations, you should decide whether to reduce the size of the cache. In this case, you may make these decisions:

- If memory is scarce, it may be wise to remove 25 buffers from the cache. The buffers 76 to 100 contribute relatively few cache hits compared to the total cache hits contributed by the entire cache. Removing 25 buffers will not significantly reduce cache performance, and the leftover memory may be better used by other Oracle memory structures.
- It is unwise to remove more than 25 buffers from the cache. For example, removing 50 buffers would reduce cache performance significantly. The cache hits contributed by these buffers is a significant portion of the total cache hits.

Reallocating Memory

After resizing your Oracle memory structures, re-evaluate the performance of the library cache, the data dictionary cache, and the buffer cache. If you have reduced the memory consumption of any one of these structures, you may want to allocate more memory to another structure. For example, if you have reduced the size of your buffer cache, you may now want to take advantage of the additional available memory by using it for the library cache.

Tune your operating system again. Resizing Oracle memory structures may have changed Oracle memory requirements. In particular, be sure paging and swapping is not excessive. For example, if the size of the data dictionary cache or the buffer cache has increased, the SGA may be too large to fit into main memory. In this case, the SGA could be paged or swapped.

While reallocating memory, you may determine that the optimum size of Oracle memory structures requires more memory than your operating system can provide. In this case, you may improve performance even further by adding more memory to your computer.

Tuning I/O

This chapter presents tuning I/O. This chapter teaches you how to avoid I/O bottlenecks that could prevent Oracle from performing at its maximum potential. This chapter covers the following topics:

- the importance of tuning I/O
- how to reduce disk contention
- properly allocating space in data blocks
- avoiding dynamic space management

The Importance of Tuning I/O

The performance of many software applications is inherently limited by disk I/O. Often, CPU activity must be suspended while I/O activity completes. Such an application is said to be “I/O bound”. Oracle is designed so that performance need not be limited by I/O.

Tuning I/O can help performance if a disk containing database files is operating at its capacity. However, tuning I/O cannot help performance in “CPU bound” cases, or cases in which your computer’s CPUs are operating at their capacity.

It is important to tune I/O after following the memory allocation recommendations presented in Chapter 8, “Tuning Memory Allocation”. Chapter 8 shows you how to allocate memory to reduce I/O to a minimum. After reaching this minimum, follow the instructions in this chapter to perform the necessary I/O as efficiently as possible.

Reducing Disk Contention

In this section you will learn how to reduce disk contention. The following issues are discussed:

- what disk contention is
- how to monitor disk activity
- how to reduce disk activity

What Is Disk Contention?

Disk contention occurs when multiple processes try to access the same disk simultaneously. Most disks have limits on both the number of accesses and the amount of data they can transfer per second. When these limits are reached, processes may have to wait to access the disk.

Monitoring Disk Activity

Disk activity is reflected by

- Oracle file I/O statistics
- operating system statistics

Oracle compiles Oracle file I/O statistics that reflect disk access to database files. Your operating system may also keep statistics for disk access to all files.



OSDoc

Additional Information: For more information about monitoring and tuning I/O, refer to your Oracle operating system-specific documentation.

Monitoring Oracle Disk Activity

Examine disk access to database files through the dynamic performance table V\$FILESTAT. By default, this table is only available to the user SYS and to users granted SELECT ANY TABLE system privilege, such as SYSTEM. These column values reflect the number of disk accesses for each datafile:

- PHYRDS The value of this column is the number of reads from each database file.
- PHYWRTS The value of this column is the number of writes to each database file.

Monitor these values over some period of time while your application is running with this query:

```
SELECT name, phylds, phywrts
FROM v$datafile df, v$filestat fs
WHERE df.file# = fs.file#;
```

This query also retrieves the name of each datafile from the dynamic performance table V\$DATAFILE. The output of this query might look like this:

NAME	PHYRDS	PHYWRTS
/oracle/ora70/dbs/ora_system.dbf	7679	2735
/oracle/ora70/dbs/ora_temp.dbf	32	546

The PHYRDS and PHYWRTS columns of V\$FILESTAT can also be obtained through SNMP.

The total I/O for a single disk is the sum of PHYRDS and PHYWRTS for all the database files managed by the Oracle instance on that disk. Determine this value for each of your disks. Also determine the rate at which I/O occurs for each disk by dividing the total I/O by the interval of time over which the statistics were collected.

Monitoring Operating System Disk Activity

Disks holding datafiles and redo log files may also hold files that are not related to Oracle. Access to such files can only be monitored through operating system facilities rather than through the V\$FILESTAT table. Such facilities may be documented in either the Oracle installation or user's guide for your operating system or your operating system documentation.

Use your operating system facilities to examine the total I/O to your disks. Try to reduce any heavy access to disks that contain database files.

Distributing I/O

Consider the statistics in the V\$FILESTAT table and your operating system facilities. Consult your hardware documentation to determine the limits on the capacity of your disks. Any disks operating at or near full capacity are potential sites for disk contention. For example, 40 or more I/Os per second is excessive for most disks on VMS or UNIX operating systems.

To reduce the activity on an overloaded disk, move one or more of its heavily accessed files to a less active disk. Apply this principle to each of your disks until they all have roughly the same amount of I/O. This is referred to as *distributing I/O*.

This section discusses guidelines for distributing I/O:

- Separate datafiles and redo log files on different disks.
- Separate, or “stripe”, table data on different disks.
- Separate tables and indexes on different disks.
- Reduce disk I/O not related to Oracle.

Separating Datafiles and Redo Log Files

Oracle processes constantly access datafiles and redo log files. If these files are on common disks, there is potential for disk contention.

Place each datafile on a separate disk. Multiple processes can then access different files concurrently without disk contention.

Place each set of redo log files on a separate disk with no other activity. Redo log files are written by the Log Writer process (LGWR) when a transaction is committed. Information in a redo log file is written sequentially. This sequential writing can take place much faster if there is no concurrent activity on the same disk.

Dedicating a separate disk to redo log files usually ensures that LGWR runs smoothly with no further tuning attention. Performance bottlenecks related to LGWR are rare. For information on tuning LGWR, see the section “Reducing Contention for Redo Log Buffer Latches” on page 10 – 9.

Note: *Mirroring* redo log files, or maintaining multiple copies of each redo log file on separate disks, does not slow LGWR considerably. LGWR writes to each disk in parallel and waits until each part of the parallel write is complete. Since the time required for your operating system to perform a single-disk write may vary, increasing the number of copies increases the likelihood that one of the single-disk writes in the parallel write

will take longer than average. A parallel write will not take longer than the longest possible single-disk write. There may also be some overhead associated with parallel writes on your operating system.

Dedicating separate disks and mirroring redo log files are important safety precautions. Dedication of separate disks to datafiles and redo log files ensures that both the datafiles and the redo log files cannot be lost in a single disk failure. Mirroring redo log files ensures that a single redo log file cannot be lost in a single disk failure.

"Striping" Table Data

"Striping" is the practice of dividing a large table's data into small portions and storing these portions in separate datafiles on separate disks. This permits multiple processes to access different portions of the table concurrently without disk contention. "Striping" is particularly helpful in optimizing random access to tables with many rows. Striping can either be done manually (described below), or through some operating system striping utilities.

To create a "striped" table:

1. Create a tablespace with the CREATE TABLESPACE command. Specify the datafiles in the DATAFILE clause. Each of the files should be on a different disk.

```
CREATE TABLESPACE stripedtablespace
DATAFILE 'file_on_disk_1' SIZE 500K,
        'file_on_disk_2' SIZE 500K,
        'file_on_disk_3' SIZE 500K,
        'file_on_disk_4' SIZE 500K,
        'file_on_disk_5' SIZE 500K;
```

2. Then create the table with the CREATE TABLE command. Specify the newly created tablespace in the TABLESPACE clause.

Also specify the size of the table extents in the STORAGE clause. Store each extent in a separate datafile. The table extents should be slightly smaller than the datafiles in the tablespace to allow for overhead. For example:

```
CREATE TABLE stripedtab (
  col_1  NUMBER(2),
  col_2  VARCHAR2(10) )
TABLESPACE stripedtablespace
STORAGE ( INITIAL 495K NEXT 495K
          MINEXTENTS 5 PCTINCREASE 0 );
```


Separating Tables and Indexes

These steps result in the creation of table STRIPEDTAB. STRIPEDTAB has 5 initial extents, each of size 495 kilobytes. Each extent takes up one of the datafiles named in the DATAFILE clause of the CREATE TABLESPACE statement. These files are all on separate disks. These 5 extents are all allocated immediately, since MINEXTENTS is 5. For more information on MINEXTENTS and the other storage parameters, see the *Oracle7 Server Administrator's Guide*.

Place frequently accessed database structures in separate datafiles on separate disks. To do this, you must know which of your database structures are used often. For example, separate an often used table from its index. This separation distributes the I/O to the table and index across separate disks.

Follow these steps to separate a table and its index:

1. Create a tablespace with the CREATE TABLESPACE command. Specify the datafile in the DATAFILE clause:

```
CREATE TABLESPACE tabspace_1
  DATAFILE 'file_on_disk_1';
```

2. Create the table with the CREATE TABLE command. Specify the tablespace in the TABLESPACE clause:

```
CREATE TABLE tab_1 (
  col_1  NUMBER(2),
  col_2  VARCHAR2(10) )
TABLESPACE tabspace_1;
```

3. Create another tablespace. Specify a datafile on another disk:

```
CREATE TABLESPACE tabspace_2
  DATAFILE 'file_on_disk_2';
```

4. Create the index. Specify the new tablespace:

```
CREATE INDEX ind_1 ON tab_1 (col_1)
  TABLESPACE tabspace_2;
```

These steps result in the creation of table TAB_1 in the file FILE_ON_DISK_1 and the creation of index IND_1 in the file FILE_ON_DISK_2.

Eliminating Other Disk I/O

If possible, eliminate I/O not related to Oracle on disks that contain database files. This measure is especially helpful in optimizing access to redo log files. Not only does this reduce disk contention, it also allows you to monitor all activity on such disks through the dynamic performance table V\$FILESTAT.

Modifying the SQL.BSQ File

The SQL.BSQ file is run when you issue the CREATE DATABASE statement. This file contains the actual table definitions that make up the Oracle Server. The views that you use as a DBA are based on these tables. Oracle Corporation recommends that users strictly limit their modifications to SQL.BSQ.

- If necessary, you can increase the value of the following storage parameters: INITIAL, NEXT, MINEXTENTS, MAXEXTENTS, PCTINCREASE, FREELISTS, FREELIST GROUPS, and OPTIMAL. See *Oracle7 Server SQL Reference* for complete information about these parameters.
- With the exception of PCTINCREASE, you should not decrease the setting of a storage parameter to a value below the default. (If the value of MAXEXTENTS is large, you can make PCTINCREASE small—or even 0.)
- No other changes to SQL.BSQ are supported. In particular, you should not add, drop, or rename a column.

Note: Internal data dictionary tables may be added, deleted, or changed from release to release. For this reason, user modifications will not be carried forward when the dictionary is migrated to later releases.

Allocating Space in Data Blocks

Table data in the database is stored in data blocks. In this section, you will learn how to allocate space within data blocks for best performance. The following issues are discussed in this section:

- how to control data storage
- how to store data most efficiently based on your application

Migrated and Chained Rows

If an UPDATE statement increases the amount of data in a row so that the row no longer fits in its data block, Oracle tries to find another block with enough free space to hold the entire row. If such a block is available, Oracle moves the entire row to the new block. This is called *migrating* a row. If the row is too large to fit into any available block, Oracle splits the row into multiple pieces and stores each piece in a separate block. This is called *chaining* a row. Rows can also be chained when they are inserted.

Dynamic space management, especially migration and chaining, is detrimental to performance:

- UPDATE statements that cause migration and chaining perform poorly.
- Queries that select migrated or chained rows must perform more I/O.

You can identify migrated and chained rows in a table or cluster by using the ANALYZE command with the LIST CHAINED ROWS option. This command collects information about each migrated or chained row and places this information into a specified output table. The definition of a sample output table named CHAINED_ROWS appears in a SQL script available on your distribution media. The common name of this script is UTLCHAIN.SQL, although its exact name and location may vary depending on your operating system. Your output table must have the same column names, datatypes, and sizes as the CHAINED_ROWS table.

To reduce migrated and chained rows in an existing table, follow these steps:

1. Use the ANALYZE command to collect information about migrated and chained rows. For example:

```
ANALYZE TABLE order_hist LIST CHAINED ROWS;
```


2. Query the output table:

```
SELECT *
FROM chained_rows
WHERE table_name = 'ORDER_HIST';
```

OWNER_NAME	TABLE_NAME	CLUST...	HEAD_ROWID	TIMESTAMP
		...		
SCOTT	ORDER_HIST	...	0000186A.0003.0001	04-AUG-92
SCOTT	ORDER_HIST	...	0000186A.0002.0001	04-AUG-92
SCOTT	ORDER_HIST	...	0000186A.0001.0001	04-AUG-92

3. If the output table shows that you have many migrated or chained rows, follow these steps to eliminate migrated rows:

3.1 Create an intermediate table with the same columns as the existing table to hold the migrated and chained rows:

```
CREATE TABLE int_order_hist
AS SELECT *
FROM order_hist
WHERE ROWID IN
(SELECT head_rowid
FROM chained_rows
WHERE table_name = 'ORDER_HIST');
```

3.2 Delete the migrated and chained rows from the existing table:

```
DELETE FROM order_hist
WHERE ROWID IN
(SELECT head_rowid
FROM chained_rows
WHERE table_name = 'ORDER_HIST');
```

3.3 Insert the rows of the intermediate table into the existing table:

```
INSERT INTO order_hist
SELECT *
FROM int_order_hist;
```

3.4 Drop the intermediate table:

```
DROP TABLE int_order_history;
```

4. Delete the information collected in step 1 from the output table:

```
DELETE FROM chained_rows
WHERE table_name = 'ORDER_HIST';
```

5. Use the ANALYZE command again and query the output table.

- 6. Any rows that appear in the output table are chained rows. You can eliminate chained rows by increasing your data block size. It may not be possible to avoid chaining in all situations. If your table has a LONG column or long CHAR or VARCHAR2 columns, chaining is often unavoidable.

Avoiding Dynamic Space Management

When an object such as a table or rollback segment is created, space is allocated in the database for the data. This space is called a *segment*. If subsequent database operations cause the data to grow and exceed the space allocated, Oracle extends the segment. Dynamic extension can reduce performance. This section discusses

- how to detect dynamic extension
- how to allocate enough space for your data to avoid dynamic extension
- how to avoid dynamic space management in rollback segments

Detecting Dynamic Extension

Dynamic extension causes Oracle to execute SQL statements in addition to those SQL statements issued by user processes. These SQL statements are called *recursive calls* because Oracle issues these statements itself. Recursive calls are also generated by these activities:

- misses on the data dictionary cache
- firing of database triggers
- execution of Data Definition Language statements
- execution of SQL statements within stored procedures, functions, packages, and anonymous PL/SQL blocks
- enforcement of referential integrity constraints

Examine the *recursive calls* statistic through the dynamic performance table V\$SYSSTAT. By default, this table is only available to the user SYS and to users granted the SELECT ANY TABLE system privilege, such as SYSTEM. Monitor this statistic over some period of time while your application is running with this query:

```
SELECT name, value
FROM v$sysstat
WHERE name = 'recursive calls';
```

The output of this query might look like this:

NAME	VALUE
recursive calls	626681

If Oracle continues to make an excess of recursive calls while your application is running, determine whether these recursive calls are due to one of the activities that generate recursive calls other than dynamic extension. If you determine that these recursive calls are caused by dynamic extension, you should try to reduce this extension by allocating larger extents.

Allocating Extents

Follow these steps to avoid dynamic extension:

1. Determine the maximum size of your object. For formulas to predict how much space to allow for a table, see the *Oracle7 Server Administrator's Guide*.
2. Choose storage parameter values so that Oracle allocates extents large enough to accommodate all of your data when you create the object.

Larger extents tend to benefit performance for these reasons:

- Since blocks in a single extent are contiguous, one large extent is more contiguous than multiple small extents. Oracle can read one large extent from disk with fewer multi-block reads than would be required to read many small extents.
- Segments with larger extents are less likely to be extended.

However, since large extents require more contiguous blocks, Oracle may have difficulty finding enough contiguous space to store them. To determine whether to allocate few large extents or many small extents, consider the benefits and drawbacks of each in light of your plans for the growth and use of your tables.

Automatically resizable datafiles can also cause a problem with dynamic extension. Avoid using the automatic extension, instead manually allocate more space to a datafile during times when the system is relatively inactive.

Avoiding Dynamic Space Management in Rollback Segments

The size of rollback segments can also affect performance. Rollback segment size is determined by the rollback segment's storage parameter values. Your rollback segments must be large enough to hold the rollback entries for your transactions. As with other objects, you should avoid dynamic space management in rollback segments.

Use the SET TRANSACTION command to assign transactions to rollback segments of the appropriate size based on the recommendations in the following sections. If you do not explicitly assign a rollback segment to a transaction, Oracle randomly chooses a rollback segment for it.



Warning: If you are running multiple concurrent copies of the same application, be careful not to assign the transactions for all copies to the same rollback segment. This leads to contention for that rollback segment.

Also monitor the shrinking, or dynamic deallocation, of rollback segments based on the OPTIMAL storage parameter. For information on choosing values for this parameter and monitoring rollback segment shrinking, and adjusting OPTIMAL accordingly, see the *Oracle7 Server Administrator's Guide*.

Example This statement assigns the current transaction to the rollback segment OLTP_13:

```
SET TRANSACTION USE ROLLBACK SEGMENT oltp_13
```

For Long Queries	Assign large rollback segments to transactions that modify data that is concurrently selected by long queries. Such queries may require access to rollback segments to reconstruct a read-consistent version of the modified data. These rollback segments must be large enough to hold all the rollback entries for the data while the query is running.
For Long Transactions	Assign large rollback segments to long transactions that modify large amounts of data. A large rollback segment can improve the performance of such a transaction. Such transactions generate large rollback entries. If a rollback entry does not fit into a rollback segment, Oracle extends the segment. Dynamic extension reduces performance and should be avoided whenever possible.
For OLTP Transactions	Some applications perform <i>online transaction processing</i> , or OLTP. OLTP applications are characterized by frequent concurrent transactions that each modify small amounts of data. Assign small rollback segments to OLTP transactions provided that their data is not concurrently queried. Small rollback segments are more likely to remain stored in the buffer cache where they can be accessed quickly. A typical OLTP rollback segment might have 2 extents, each approximately 10 kilobytes in size. To best avoid contention, create many rollback segments and assign each transaction to its own rollback segment.

Tuning Contention

This chapter discusses tuning contention. Contention occurs when multiple processes try to access the same resource simultaneously. Contention causes processes to wait for access to various database structures.

Oracle provides you with methods of handling contention. In this chapter, you will learn how to

- detect contention that may affect performance
- reduce contention

Topics discussed in this chapter include

- rollback segments
- processes of the multi-threaded server
- query server processes for the parallel query option
- redo log buffer latches
- LRU latch contention
- assigning processes equal priority

Reducing Contention for Rollback Segments

In this section, you will learn how to reduce contention for rollback segments. The following issues are discussed:

- how to identify contention for rollback segments
- how to create rollback segments
- how to assign rollback segments to particular transactions

Identifying Rollback Segment Contention

Contention for rollback segments is reflected by contention for buffers that contain rollback segment blocks. You can determine whether contention for rollback segments is reducing performance by using the dynamic performance table V\$WAITSTAT.

The V\$WAITSTAT contains statistics that reflect block contention. By default, this table is only available to the user SYS and to other users who have SELECT ANY TABLE system privilege, such as SYSTEM. These statistics reflect contention for different classes of block:

<i>system undo header</i>	The value of this statistic is the number of waits for buffers containing header blocks of the SYSTEM rollback segment.
<i>system undo block</i>	The value of this statistic is the number of waits for buffers containing blocks other than header blocks of the SYSTEM rollback segment.
<i>undo header</i>	The value of this statistic is the number of waits for buffers containing header blocks of rollback segments other than the SYSTEM rollback segment.
<i>undo block</i>	The value of this statistic is the number of waits for buffers containing blocks other than header blocks of rollback segments other than the SYSTEM rollback segment.

Monitor these statistics over a period of time while your application is running with this query:

```
SELECT class, count
FROM v$waitstat
WHERE class IN ('system undo header', 'system undo block',
               'undo header', 'undo block');
```


The result of this query might look like this:

CLASS	COUNT
system undo header	2089
system undo block	633
undo header	1235
undo block	942

Compare the number of waits for each class of block with the total number of requests for data over the same period of time. You can monitor the total number of requests for data over a period of time with this query:

```
SELECT SUM(value)
  FROM v$sysstat
 WHERE name IN ('db block gets', 'consistent gets');
```

The output of this query might look like this:

SUM(VALUE)
929530

The information in V\$SYSSTAT can also be obtained through SNMP.

If the number of waits for any class is greater than 1% of the total number of requests, you should consider creating more rollback segments to reduce contention.

Creating Rollback Segments

To reduce contention for buffers containing rollback segment blocks, create more rollback segments. Table 10 – 1 shows some general guidelines for choosing how many rollback segments to allocate based on the number of concurrent transactions on your database. These guidelines are appropriate for most application mixes.

Number of Current Transactions(n)	Recommended Number of Rollback Segments
n<16	4
16<=n<32	8
32<=n	n/4, but no more than 50

Table 10 – 1 Choosing a Number of Rollback Segments

Reducing Contention for Multi-Threaded Server Processes

In this section, you will learn how to reduce contention for these processes used by the Oracle's multi-threaded server architecture:

- dispatcher processes
- shared server processes

Reducing Contention for Dispatcher Processes

This section discusses these issues:

- how to identify contention for dispatcher processes
- how to add dispatcher processes

Identifying Contention for Dispatcher Processes

Contention for dispatcher processes can be reflected by either of these symptoms:

- high busy rates for existing dispatcher processes
- steady increase in waiting time for responses in the response queues of existing dispatcher processes

Examining Busy Rates for Dispatcher Processes Statistics reflecting the activity of dispatcher processes are kept in the dynamic performance table V\$DISPATCHER. By default, this table is only available to the user SYS and to other users who have SELECT ANY TABLE system privilege, such as SYSTEM. These columns reflect busy rates for dispatcher processes:

IDLE	This column shows the idle time for the dispatcher process in hundredths of seconds.
BUSY	This column shows the busy time for the dispatcher process in hundredths of seconds.

Monitor these statistics over a period of time while your application is running with this query:

```
SELECT network                                "Protocol",
       SUM(busy) / ( SUM(busy) + SUM(idle) )  "Total Busy Rate"
FROM   v$dispatcher
GROUP BY network;
```

This query returns the total busy rate for the dispatcher processes of each protocol or the percentage of time the dispatcher processes of each protocol are busy. The result of this query might look like this:

Protocol	Total Busy Rate
decnet	.004589828
tcp	.029111042

From this result, you can make these observations:

- DECnet dispatcher processes are busy nearly 0.5% of the time.
- TCP dispatcher processes are busy nearly 3% of the time.

If the dispatcher processes for a specific protocol are busy more than 50% of the time, then you may be able to improve performance for users connected to Oracle using that protocol by adding dispatcher processes.

Examining Wait Times for Dispatcher Process Response Queues Statistics reflecting the response queue activity for dispatcher processes are kept in the dynamic performance table V\$QUEUE. By default, this table is only available to the user SYS and to other users who have SELECT ANY TABLE system privilege, such as SYSTEM. These columns show wait times for responses in the queue:

WAIT	This column shows the total waiting time in hundredths of seconds for all responses that have ever been in the queue.
TOTALQ	This column shows the total number of responses that have ever been in the queue.

Monitor these statistics occasionally while your application is running with this query:

```
SELECT network      "Protocol",
       DECODE( SUM(totalq), 0, 'No Responses',
              SUM(wait)/SUM(totalq) || ' hundredths of seconds')
       "Average Wait Time per Response"
FROM v$queue q, v$dispatcher d
WHERE q.type = 'DISPATCHER'
      AND q.paddr = d.paddr
GROUP BY network;
```

This query returns the average time in hundredths of seconds that a response waits in each response queue for a dispatcher process to route it to a user process. This query uses the V\$DISPATCHER table to group the rows of the V\$QUEUE table by network protocol. The query also uses the DECODE syntax to recognize those protocols for which there have been no responses in the queue. The result of this query might look like this:

Protocol	Average Wait Time per Response
decnet	.1739130 hundredths of seconds
tcp	No Responses

From this result, you can tell that a response in the queue for DECNET dispatcher processes waits an average of 0.17 hundredths of a second and that there have been no responses in the queue for TCP dispatcher processes.

If the average wait time for a specific network protocol continues to increase steadily as your application runs, you may be able to improve performance of those user processes connected to Oracle using that protocol by adding dispatcher processes.

Adding Dispatcher Processes

You can add dispatcher processes while Oracle is running with the MTS_DISPATCHERS parameter of the ALTER SYSTEM command.

For more information on adding dispatcher processes, see the *Oracle7 Server Administrator's Guide*.

The total number of dispatcher processes across all protocols is limited by the value of the initialization parameter MTS_MAX_DISPATCHERS. You may need to increase this value before adding dispatcher processes. The default value of this parameter is 5 and the maximum value varies depending on your operating system.

Reducing Contention for Shared Server Processes

This section discusses these issues:

- how to identify contention for shared server processes
- how to increase the maximum number of shared server processes

Identifying Contention for Shared Server Processes

Contention for shared server processes can be reflected by steady increase in waiting time for requests in the request queue. Statistics reflecting the request queue activity for shared server processes are kept in the dynamic performance table V\$QUEUE. By default, this table is only available to the user SYS and to other users who have SELECT ANY TABLE system privilege, such as SYSTEM. These columns show wait times for requests in the queue:

WAIT	This column shows the total waiting time in hundredths of seconds for all requests that have ever been in the queue.
TOTALQ	This column shows the total number of requests that have ever been in the queue.

Monitor these statistics occasionally while your application is running with this query:

```
SELECT DECODE( totalq, 0, 'No Requests',
              wait/totalq || ' hundredths of seconds')
       "Average Wait Time Per Requests"
FROM v$queue
WHERE type = 'COMMON';
```

This query returns the total wait time for all requests and total number of requests for the request queue. The result of this query might look like this:

```
Average Wait Time per Request
-----
.090909 hundredths of seconds
```

From the result, you can tell that a request waits an average of 0.09 hundredths of a second in the queue before it is processed.

You can also determine how many shared server processes are currently running by issuing this query:

```
SELECT COUNT(*) "Shared Server Processes"
FROM v$shared_servers
WHERE status != 'QUIT';
```

The result of this query might look like this:

```
Shared Server Processes
-----
10
```

Adding Shared Server Processes

Since Oracle automatically adds shared server processes if the load on existing shared server processes increases drastically, you are unlikely to improve performance simply by explicitly adding more shared server processes. However, if the number of shared server processes has reached the limit established by the initialization parameter `MTS_MAX_SERVERS` and the average wait time in the requests queue is still increasing, you may be able to improve performance by increasing the `MTS_MAX_SERVERS` value. The default value of this parameter is 20 and the maximum value varies depending on your operating system. You can then either allow Oracle to automatically add shared server processes or explicitly add shared processes through one of these means:

- the `MTS_SERVERS` initialization parameter
- the `MTS_SERVERS` parameter of the `ALTER SYSTEM` command

For more information on adding shared server processes, see the *Oracle7 Server Administrator's Guide*.

Reducing Contention for Query Servers

Identifying Query Server Contention

This section describes how to detect and alleviate contention for query servers when using the parallel query option.

The V\$PQ_SYSSTAT view contains statistics that are useful for determining the appropriate number of query server processes for an instance. The statistics that are particularly useful are "Servers Busy", "Servers Idle", "Servers Started", and "Servers Shutdown".

Frequently, you will not be able to increase the maximum number of query servers for an instance because the maximum number is heavily dependent upon the capacity of your CPUs and your I/O bandwidth. However, if servers are continuously starting and shutting down, you should consider increasing the value of the parameter PARALLEL_MIN_SERVERS.

For example, if you have determined that the maximum number of concurrent query servers that your machine can manage is 100, you should set PARALLEL_MAX_SERVERS to 100. Next determine how many query servers the average query needs, and how many queries are likely to be executed concurrently. For this example, assume you will have two concurrent queries with 20 as the average degree of parallelism. Thus, at any given point in time, there could be 80 query servers busy on an instance. Thus you should set the parameter PARALLEL_MIN_SERVERS to be 80.

Now you should periodically examine V\$PQ_SYSSTAT to determine if the 80 query servers for the instance are actually busy. To determine if the instance's query servers are active, issue the following query:

```
SELECT * FROM V$PQ_SYSSTAT
      WHERE statistic = "Servers Busy";
```

STATISTIC	VALUE
Servers Busy	70

Reducing Query Server Contention

If you find that there are typically fewer than PARALLEL_MIN_SERVERS busy at any given time, your idle query servers are additional system overhead that is not being used. You should then consider decreasing the value of the parameter PARALLEL_MIN_SERVERS. If you find that there are typically more query servers active than the value of PARALLEL_MIN_SERVERS and the "Servers Started" statistic is continuously growing, then you should consider increasing the value of the parameter PARALLEL_MIN_SERVERS.

Reducing Contention for Redo Log Buffer Latches

Contention for redo log buffer access rarely inhibits database performance. However, Oracle provides you with methods to monitor and reduce any latch contention that does occur. This section explains

- how to detect and reduce contention for space in the redo log buffer
- how to detect contention for latches
- how to reduce contention for latches

Space in the Redo Log Buffer

When LGWR writes redo entries from the redo log buffer to a redo log file, user processes can then copy new entries over the entries that have been written to disk. LGWR normally writes fast enough to ensure that space is always available in the buffer for new entries, even when access to the redo log is heavy.

The statistic *redo log space requests* reflects the number of times a user process waits for space in the redo log buffer. This statistic is available through the dynamic performance table V\$SYSSTAT. By default, this table is only available to the user SYS and to users granted SELECT ANY TABLE system privilege, such as SYSTEM. Monitor this statistic over a period of time while your application is running with this query:

```
SELECT name, value
FROM v$sysstat
WHERE name = 'redo log space requests';
```

The information in V\$SYSSTAT can also be obtained through SNMP.

The value of *redo log space requests* should be near 0. If this value increments consistently, processes have had to wait for space in the buffer. In this case, increase the size of the redo log buffer. The size of the redo log buffer is determined by the initialization parameter LOG_BUFFER. The value of this parameter, expressed in bytes, must be a multiple of DB_BLOCK_SIZE.

Redo Log Buffer Latches

Access to the redo log buffer is regulated by latches. Two types of latches control access to the redo log buffer:

- the redo allocation latch
- redo copy latches

The Redo Allocation Latch The *redo allocation latch* controls the allocation of space for redo entries in the redo log buffer. To allocate space in the buffer, an Oracle user process must obtain the redo allocation latch. Since there is only one redo allocation latch, only one user process can allocate space in the buffer at a time. The single redo allocation latch enforces the sequential nature of the entries in the buffer.

After allocating space for a redo entry, the user process may copy the entry into the buffer while holding the redo allocation latch. Such a copy is referred to as “copying on the redo allocation latch”. A process may only copy on the redo allocation latch if the redo entry is smaller than a threshold size. After copying on the redo allocation latch, the user process releases the latch.

The maximum size of a redo entry that can be copied on the redo allocation latch is specified by the initialization parameter `LOG_SMALL_ENTRY_MAX_SIZE`. The value of this parameter is expressed in bytes. The minimum, maximum, and default values vary depending on your operating system.

Redo Copy Latches

If the redo entry is too large to copy on the redo allocation latch, the user process must obtain a *redo copy latch* before copying the entry into the buffer. While holding a redo copy latch, the user process copies the redo entry into its allocated space in the buffer and then releases the redo copy latch.

If your computer has multiple CPUs, your redo log buffer can have multiple redo copy latches. Multiple redo copy latches allow multiple processes to copy entries to the redo log buffer concurrently. The number of redo copy latches is determined by the initialization parameter `LOG_SIMULTANEOUS_COPIES`. The default value of `LOG_SIMULTANEOUS_COPIES` is the number of CPUs available to your Oracle instance.

On single-CPU computers, there should be no redo copy latches since only one process can be active at once. In this case, all redo entries are copied on the redo allocation latch, regardless of size.

Examining Redo Log Activity

Heavy access to the redo log buffer can result in contention for redo log buffer latches. Latch contention can reduce performance. Oracle collects statistics for the activity of all latches and stores them in the dynamic performance table `V$LATCH`. By default, this table is only available to the user `SYS` and to other users who have `SELECT ANY TABLE` system privilege, such as `SYSTEM`.

Each row in the V\$LATCH table contains statistics for a different type of latch. The columns of the table reflect activity for different types of latch requests. The distinction between these types of requests is whether the requesting process continues to request a latch if it is unavailable:

willing-to-wait	If the latch requested with a willing-to-wait request is not available, the requesting process waits a short time and requests the latch again. The process continues waiting and requesting until the latch is available.
immediate	If the latch requested with an immediate request is not available, the requesting process does not wait, but continues processing.

These columns of the V\$LATCH table reflect willing-to-wait requests:

GETS	This column shows the number of successful willing-to-wait requests for a latch.
MISSES	This column shows the number of times an initial willing-to-wait request was unsuccessful.
SLEEPS	This column shows the number of times a process waited and requested a latch after an initial willing-to-wait request.

For example, consider the case in which a process makes a willing-to-wait request for a latch that is unavailable. The process waits and requests the latch again and the latch is still unavailable. The process waits and requests the latch a third time and acquires the latch. This activity increments the statistics in these ways:

- The GETS value increases by one, since one request for the latch (the third request) was successful.
- The MISSES value increases by one, since the initial request for the latch resulted in waiting.
- The SLEEPS value increases by two, since the process waited for the latch twice, once after the initial request and again after the second request.

These columns of the V\$LATCH table reflect immediate requests:

IMMEDIATE GETS	This column shows the number of successful immediate requests for each latch.
IMMEDIATE MISSES	This column shows the number of unsuccessful immediate requests for each latch.

Monitor the statistics for the redo allocation latch and the redo copy latches over a period of time with this query:

```
SELECT ln.name, gets, misses, immediate_gets, immediate_misses
FROM v$latch l, v$latchname ln
WHERE ln.name IN ('redo allocation', 'redo copy')
AND ln.latch# = l.latch#;
```

The output of this query might look like this:

NAME	GETS	MISSES	IMMEDIATE_GETS	IMMEDIATE_MISSES
redo allo...	252867	83	0	0
redo copy	0	0	22830	0

From the output of the query, calculate the wait ratio for each type of request.

Contention for a latch may be affecting performance if either of these conditions is true:

- if the ratio of MISSES to GETS exceeds 1%
- if the ratio of IMMEDIATE_MISSES to the sum of IMMEDIATE_GETS and IMMEDIATE_MISSES exceeds 1%

If either of these conditions is true for a latch, try to reduce contention for that latch.

These contention thresholds are appropriate for most operating systems, though some computers with many CPUs may be able to tolerate more contention without performance reduction.

Reducing Latch Contention

Most cases of latch contention occur when two or more Oracle processes concurrently attempt to obtain the same latch. Latch contention rarely occurs on single-CPU computers where only a single process can be active at once.

Reducing Contention for the Redo Allocation Latch

To reduce contention for the redo allocation latch, you should minimize the time that any single process holds the latch. To reduce this time, reduce copying on the redo allocation latch. Decreasing the value of the `LOG_SMALL_ENTRY_MAX_SIZE` initialization parameter reduces the number and size of redo entries copied on the redo allocation latch.

Reducing Contention for Redo Copy Latches

On multiple-CPU computers, multiple redo copy latches allow multiple processes to copy entries to the redo log buffer concurrently. The default value of `LOG_SIMULTANEOUS_COPIES` is the number of CPUs available to your Oracle instance.

If you observe contention for redo copy latches, add more latches. To increase the number of redo copy latches, increase the value of `LOG_SIMULTANEOUS_COPIES`. It can help to have up to twice as many redo copy latches as CPUs available to your Oracle instance.

Reducing LRU Latch Contention

Contention for the LRU latch can impede performance on symmetric multiprocessor (SMP) machines with a large number of CPUs. The LRU latch controls the replacement of buffers in the buffer cache. For SMP systems, Oracle automatically sets the number of LRU latches to be one half the number of CPUs on the system. For non-SMP systems, one LRU latch is sufficient.

You can specify the number of LRU latches on your system with the initialization parameter `DB_BLOCK_LRU_LATCHES`. This parameter sets the maximum value for the desired number of LRU latches. Each LRU latch will control a set of buffers and Oracle balances allocation of replacement buffers among the sets. To select the appropriate value for `DB_BLOCK_LRU_LATCHES`, consider the following:

- The maximum number of latches is twice the number of CPUs in the system. For example, `DB_BLOCK_LRU_LATCHES` can range from $1 \dots 2 \times \text{CPUs}$.
- A latch should have no less than 50 buffers in its set (for small buffer caches there is no added value if you select a larger number of sets). The size of the buffer cache determines a maximum boundary condition on the number of sets.
- Do not create multiple latches when Oracle runs in *single process* mode. Oracle automatically uses only one LRU latch in single process mode.

- If your workload on the instance is large, then you should have a higher number of latches. For example, if you have 32 CPUs in your system, choose a number between half the number of CPUs (16) and actual number of CPUs (32) in your system.

Note: You cannot dynamically change the number of sets during the lifetime of the instance.

Assigning All Oracle Processes Equal Priority

Many processes are involved in the operation of Oracle. These processes all access the shared memory resources in the SGA.

Be sure that all Oracle processes, both background processes and user processes, have the same process priority. When you install Oracle, all background processes are given the default priority for your operating system. You should not change the priorities of background processes. You should also be sure that all user processes have the default operating system priority.

Assigning different priorities to Oracle processes may exacerbate the effects of contention. Your operating system may not grant processing time to a low priority process if a high priority process also requests processing time. If a high priority process needs access to a memory resource held by a low priority process, the high priority process may wait indefinitely for the low priority process to obtain the CPU, process, and release the resource.

Additional Tuning Considerations

This chapter discusses additional tuning measures you can take. Usually, these measures improve performance only in very special cases. The topics covered by this chapter include

- sorts
- indexes
- free lists
- checkpoints

Tuning Sorts

Some applications cause Oracle to sort data. This section tells you

- how to allocate memory to optimize your sorts
- how to avoid sorting when creating indexes

Allocating Memory for Sort Areas

The default sort area size is adequate to hold all the data for most sorts. However, if your application often performs large sorts on data that does not fit into the sort area, then you may want to increase the sort area size. Large sorts can be caused by any SQL statement that performs a sort that operates on many rows. SQL statements that perform sorts are listed in Chapter 9, "Memory Structures and Processes", of *Oracle7 Server Concepts*.

Recognizing Large Sorts

Oracle collects statistics that reflect sort activity and stores them in the dynamic performance table V\$SYSSTAT. By default, this table is only available to the user SYS and to users granted the SELECT ANY TABLE system privilege. These statistics reflect sort behavior:

<i>sorts(memory)</i>	The value of this statistic is the number of sorts small enough to be performed entirely in sort areas without I/O to temporary segments on disk.
<i>sorts(disk)</i>	The value of this statistic is the number of sorts too large to be performed entirely in the sort area requiring I/O to temporary segments on disk.

Monitor these statistics over a period of time while your application is running with this query:

```
SELECT name, value
FROM v$sysstat
WHERE name IN ('sorts(memory)', 'sorts(disk)');
```

The output of this query might look like this:

NAME	VALUE
<i>sorts(memory)</i>	965
<i>sorts(disk)</i>	8

The information in V\$SYSSTAT can also be obtained through SNMP.

Increasing Sort Area Size

If a significant number of sorts require disk I/O to temporary segments, then your application's performance may benefit from increasing the size of the sort area. In this case, increase the value of the initialization parameter SORT_AREA_SIZE. The maximum value of this parameter varies depending on your operating system.

Performance Benefits of Large Sort Areas	Increasing the size of the sort area increases the size of each run and decreases the total number of runs. Reducing the total number of runs may reduce the number of merges Oracle must perform to obtain the final sorted result.
Performance Tradeoffs for Large Sort Areas	<p>Increasing the size of the sort area causes each Oracle process that sorts to allocate more memory. This increase reduces the amount of memory available for private SQL and PL/SQL areas. It can also affect operating system memory allocation and may induce paging and swapping. Before increasing the size of the sort area, be sure enough free memory is available on your operating system to accommodate a larger sort area.</p> <p>If you increase the size of your sort area, you may consider decreasing the retained size of the sort area, or the size to which Oracle reduces the sort area if its data is not expected to be referenced soon. To decrease the retained size of the sort area, decrease the value of the initialization parameter SORT_AREA_RETAINED_SIZE. A smaller retained sort area reduces memory usage but causes additional I/O to write and read data to and from temporary segments on disk.</p>
Optimizing Sort Performance with TEMPORARY Tablespaces	<p>You can optimize sort performance of sorts by specifying a tablespace as TEMPORARY upon creation (or subsequently altering that tablespace) and performing the sort in that tablespace. Normally, a sort may require many space allocation calls to allocate and deallocate temporary segments. If a tablespace is specified as TEMPORARY, one sort segment in that tablespace is cached and used for each instance requesting a sort operation. This scheme bypasses the normal space allocation mechanism and can greatly improve performance of medium-sized sorts that cannot be done completely in memory.</p> <p>To specify a tablespace as temporary, use the TEMPORARY keyword of the CREATE TABLE or ALTER TABLE commands. TEMPORARY cannot be used with tablespaces that contain permanent objects (such as tables or rollback segments). See the <i>Oracle7 Server SQL Reference</i> chapter for more information about the syntax of the CREATE TABLE and ALTER TABLE commands.</p>
Avoiding Sorts	<p>One cause of sorting is the creation of indexes. Creating an index for a table involves sorting all the rows in the table based on the values of the indexed column or columns.</p> <p>Oracle also allows you to create indexes without sorting. If the rows in the table are loaded in ascending order, you can create the index faster without sorting.</p>

The NOSORT Option

To create an index without sorting, load the rows into the table in ascending order of the indexed column values. Your operating system may provide you with a sorting utility to sort the rows before you load them.

When you create the index, use the NOSORT option on the CREATE INDEX command. For example, this CREATE INDEX statement creates the index EMP_INDEX on the ENAME column of the EMP table without sorting the rows in the EMP table:

```
CREATE INDEX emp_index  
ON emp(ename)  
NOSORT;
```

Note: Specifying NOSORT in a CREATE INDEX statement negates the use of PARALLEL INDEX CREATE, even if PARALLEL (DEGREE *n*) is specified.

Choosing When to Use the NOSORT Option

Presorting your data and loading it in order may not always be the fastest way to load a table. If you have a multiple-CPU computer, you may be able to load data faster using multiple processors in parallel, each processor loading a different portion of the data. To take advantage of parallel processing, load the data without sorting it first. Then create the index without the NOSORT option.

On the other hand, if you have a single-CPU computer, you should sort your data before loading, if possible. Then you should create the index with the NOSORT option.

GROUP BY NOSORT

Sorting can be avoided when performing a GROUP BY operation when it is known that the input data is already ordered so that all rows in each group are clumped together. This may be the case, for example, if the rows are being retrieved from an index that matches the grouped columns, or if a sort-merge join produces the rows in the right order. This capability is similar to the capability to avoid ORDER BY sorts in the same circumstances. When no sort takes place, the EXPLAIN PLAN output indicates GROUP BY NOSORT.

You must set the V733_PLANS_ENABLED initialization parameter to TRUE for GROUP BY NOSORT to be available.

SORT_DIRECT_ WRITES parameter

If memory and temporary space are abundant on your system, and you perform many large sorts to disk, you can set the initialization parameter `SORT_DIRECT_WRITES` to increase sort performance. When this parameter is set to `TRUE`, each sort will allocate several large buffers in memory for direct disk I/O. You can set the initialization parameters `SORT_WRITE_BUFFERS` and `SORT_WRITE_BUFFER_SIZE` to control the number and size of these buffers. The sort will write an entire buffer for each I/O operation. The Oracle process performing the sort writes the sort data directly to the disk, bypassing the buffer cache.

The default value of `SORT_DIRECT_WRITES` is `AUTO`. When the parameter is unspecified or set to `AUTO`, Oracle automatically allocates direct write buffers if the `SORT_AREA_SIZE` is at least ten times the minimum direct write buffer configuration.

The memory for the direct write buffers is subtracted from the sort area, so the total amount of memory used for each sort is still `SORT_AREA_SIZE`. Setting `SORT_WRITE_BUFFERS` and `SORT_WRITE_BUFFER_SIZE` has no effect when `SORT_DIRECT_WRITES` is `AUTO`.

Performance Tradeoffs of Direct Disk I/O for Sorts

Setting `SORT_DIRECT_WRITES` to `TRUE` causes each Oracle process that sorts to allocate memory in addition to that already allocated for the sort area. The additional memory allocated is calculated as follows:

`SORT_WRITE_BUFFERS * SORT_WRITE_BUFFER_SIZE`

The minimum direct write configuration on most platforms is two 32K buffers (2 * 32K), so direct write is generally allocated only if the sort area is 640K or greater. With a sort area smaller than this, direct write will not be performed.

Ensure that your operating system has enough free memory available to accommodate this increase. Also, sorts that use direct writes will tend to consume more temporary segment space on disk.

One way to avoid increasing memory usage is to decrease the sort area by the amount of memory allocated for direct writes. Note that reducing the sort area may increase the number of sorts to disk, which will decrease overall performance. A good rule of thumb is that the total memory allocated for direct write buffers should be less than one-tenth of the memory allocated for the sort area. If the minimum configuration of the direct write buffers is greater than one-tenth of your sort area, then you should not trade sort area for direct write buffers.

Recreating an Index

You may wish to recreate an index in order to compact it and clean up fragmented space, or to change the index's storage characteristics. When creating a new index which is a subset of an existing index, or when rebuilding an existing index with new storage characteristics, Oracle uses the existing index instead of the base table to improve performance.

Consider, for example, a table named CUST with columns NAME, CUSTID, PHONE, ADDR, BALANCE, and an index named I_CUST_CUSTINFO on columns NAME, CUSTID and BALANCE of the table. To create a new index named I_CUST_CUSTNO on columns NAME and CUSTID, you would enter:

```
CREATE INDEX I_CUST_CUSTNO ON CUST(NAME,CUSTID)
```

Oracle will automatically use the existing index (I_CUST_CUSTINFO) to create the new index rather than accessing the entire table. Note that the syntax used is the same as if the index I_CUST_CUSTINFO did not exist.

Similarly, if you have an index on the EMPNO and MGR columns of the EMP table, and you want to change the storage characteristics of that composite index, Oracle can use the existing index to create the new index.

Use the ALTER INDEX REBUILD command to change the storage characteristics of an existing index. The REBUILD uses the existing index as the basis for the new index. All index storage commands are supported, such as STORAGE (for extent allocation), TABLESPACE (to move the index to a new tablespace), and INITRANS (to change the initial number of entries). See *Oracle7 Server SQL Reference* for more information about the CREATE INDEX and ALTER INDEX commands.

Reducing Free List Contention

Free list contention can reduce the performance of some applications. This section tells you

- how to identify contention for free lists
- how to increase the number of free lists

Identifying Free List Contention

Contention for free lists is reflected by contention for free data blocks in the buffer cache. You can determine whether contention for free lists is reducing performance by querying the dynamic performance table V\$WAITSTAT.

The V\$WAITSTAT table contains block contention statistics. By default, this table is only available to the user SYS and to other users who have SELECT ANY TABLE system privilege, such as SYSTEM. The free list statistic reflects contention for free blocks. Monitor this statistic over a period of time while your application is running with this query:

```
SELECT class, count
FROM v$waitstat
WHERE class = 'free list';
```

The result of this query might look like this:

CLASS	COUNT
free list	459

Compare the number of waits for free blocks with the total number of requests for data over the same period of time. You can monitor the total number of requests for data over a period of time with this query:

```
SELECT SUM(value)
FROM v$sysstat
WHERE name IN ('db block gets', 'consistent gets');
```

The output of this query might look like this:

SUM(VALUE)
929530

The information in V\$SYSSTAT can also be obtained through SNMP.

If the number of waits for free blocks is greater than 1% of the total number of requests, you should consider adding more free lists to reduce contention.

Adding More Free Lists

To reduce contention for the free lists of a table, re-create the table with a larger value for the FREELISTS storage parameter. Increasing the value of this parameter to the number of Oracle processes that concurrently insert data into the table may benefit performance for the INSERT statements.

Re-creating the table may simply involve dropping and creating it again. However, you may want to use one of these means instead:

- Re-create the table by selecting data from the old table into a new table, dropping the old table, and renaming the new one.
- Use Import and Export to export the table, drop the table, and import the table. This measure avoids consuming space by creating a temporary table.

Tuning Checkpoints

A checkpoint is an operation that Oracle performs automatically. Checkpoints can momentarily reduce performance. This section tells you

- how checkpoints affect performance
- how to choose the frequency of checkpoints
- how to reduce the performance impact of a checkpoint

How Checkpoints Affect Performance

Checkpoints affect

- recovery time performance
- runtime performance

Recovery Time Performance

Frequent checkpoints can reduce recovery time in the event of an instance failure. If checkpoints are relatively frequent, then relatively few changes to the database are made between checkpoints. In this case, relatively few changes must be rolled forward for recovery.

Runtime Performance

A checkpoint can momentarily reduce runtime performance for these reasons:

- Checkpoints cause DBWR to perform I/O.
- If CKPT is not enabled, checkpoints cause LGWR to update data files and may momentarily prevent LGWR from writing redo entries.

Choosing Checkpoint Frequency

The overhead associated with a checkpoint is usually small and only affects performance while Oracle performs the checkpoint.

You should choose a checkpoint frequency based on your performance concerns. If you are more concerned with efficient runtime performance than recovery time, choose a lower checkpoint frequency. If you are more concerned with fast recovery time than runtime performance, choose a higher checkpoint frequency.

Because checkpoints on log switches are necessary for redo log maintenance, you cannot eliminate checkpoints entirely. However, you can reduce checkpoint frequency to a minimum by setting these parameters:

- Set the value of the initialization parameter `LOG_CHECKPOINT_INTERVAL` larger than the size of your largest redo log file.
- Set the value of the initialization parameter `LOG_CHECKPOINT_TIMEOUT` to 0. This value eliminates time-based checkpoints.

Such settings eliminate all checkpoints except those that occur on log switches.

You can further reduce checkpoints by reducing the frequency of log switches. To reduce log switches, increase the size of your redo log files so that the files do not fill as quickly.

Reducing the Performance Impact of a Checkpoint

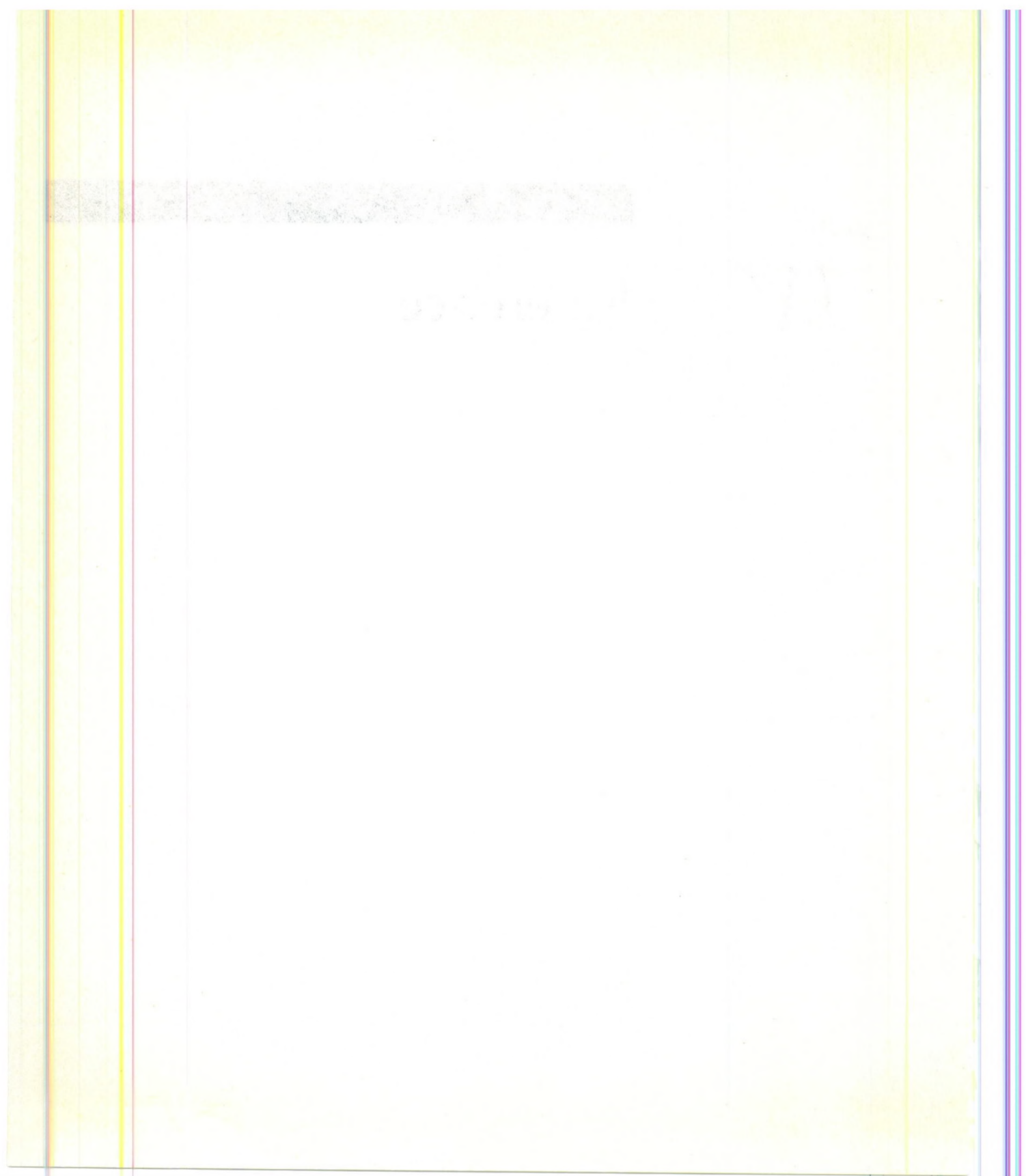
You may notice a momentary drop in performance as a checkpoint completes. This drop may be due to an accumulation of redo entries in the redo log buffer. LGWR may be too busy updating data file headers to write these entries to a redo log file. In this case, you can reduce the performance impact of a checkpoint by enabling the Checkpoint process (CKPT).

CKPT updates datafile headers when a checkpoint occurs, leaving LGWR free to write redo entries. To enable CKPT, set the value of the initialization parameter `CHECKPOINT_PROCESS` to `TRUE`. To disable CKPT, set this value to `FALSE`. CKPT is disabled by default. Before enabling CKPT, be sure that your operating system can support an additional process. You may also need to increase the value of the initialization parameter `PROCESSES`.

PART

IV

Reference



A

Performance Diagnostic Tools

The primary tool for monitoring the performance of Oracle is the collection of dynamic performance views. These views have names beginning with "V\$" and are documented in parts of this manual that describe how they can be used for monitoring performance. All dynamic performance tables are listed in the *Oracle7 Server Reference*.

These views and additional performance diagnostic tools can help you monitor and tune applications that run against the Oracle Server.

The performance tools described in this appendix are

- dynamic performance views
- the SQL trace facility
- the EXPLAIN PLAN command
- the Oracle Trace facility

The output of the EXPLAIN PLAN command reflects the behavior of the Oracle optimizer. Because the optimizer is likely to evolve between releases of the Oracle Server, output from the EXPLAIN PLAN command will also evolve. Similarly, the SQL trace facility and TKPROF program are also subject to change in future releases of the Oracle Server. Such changes will be documented in future versions of Oracle manuals.

Dynamic Performance Views

Identifying Performance Bottlenecks

The V\$SESSION_WAIT view lists the events that cause all user and system sessions to wait. Querying this view is a good way to get a quick look at possible performance bottlenecks. This view lists what each session is currently waiting for or what the session last waited for. The view gives a wait time in hundredths of seconds for each associated session wait event. The view also has three auxiliary columns (named P1, P2, and P3), which may contain additional information for each event.

Certain events described in this table are more meaningful than others; you need to collect historical data to determine if the wait times are unusually slow for certain events. The following example illustrates the contents of V\$SESSION_WAIT:

```
SQL> SELECT sid, event, wait_time
2      FROM v$session_wait
3      ORDER BY wait_time, event;
```

SID	EVENT	WAIT_TIME
9	Null event	0
174	client message	0
164	db file sequential read	0
6	rdbms ipc message	0
5	smon timer	0
108	db file sequential read	1
234	db file sequential read	1
63	log file sync	1
33	log file sync	1
201	virtual circuit status	1
89	db file sequential read	2
60	db file sequential read	2
19	log file sync	2
166	log file sync	2
233	db file sequential read	3
226	db file sequential read	3
170	log file sync	3
130	db file sequential read	4
95	db file sequential read	11
1	pmon timer	300
205	latch free	4294967295
207	latch free	4294967295
209	latch free	4294967295
215	latch free	4294967295
293	latch free	4294967295
294	latch free	4294967295
117	log file sync	4294967295
129	log file sync	4294967295
22	virtual circuit status	4294967295

Wait time equal to zero indicates that the session is currently waiting for an event. Wait time greater than zero is the time waited for the last event, in hundredths of a second. The unusually large wait times for the last several events signify that the wait time for the last event was less than one hundredth of a second (the huge number is actually an unsigned representation of negative one (-1)).

This information is only part of the picture. You need the other columns in V\$SESSION_WAIT as well as information in the other dynamic performance views before you can determine the cause of the performance problem. The previous query does, however, point you in the right direction. We can see from this example that the redo log files

and certain latches could be causing response time problems. The next section describes how to diagnose the cause of the problem.

The following table describes some of the more important events in V\$SESSION_WAIT and where in this manual those types of problems are addressed. Use this table to locate the sections on troubleshooting the specific performance problems you encounter.

Event	Possible Cause	Corrective Action/ See Page
free buffer waits	DBWR not writing frequently enough	increase number of checkpoints, on page 11 – 8
latch free	contention for latches	dependent upon the latch, on page 10 – 9
buffer busy waits	I/O contention, Parallel Server contention for data blocks	tune I/O and distribute data effectively, <i>Oracle Parallel Server Concepts & Administration</i>
db file sequential read	I/O contention	tune I/O and distribute data effectively, chapter 9 improperly tuned SQL statements, chapter 7
db file scattered read	too many table scans	tune I/O and distribute data effectively, chapter 9 improperly tuned SQL statements, chapter 7
db file parallel write	not checkpointing frequently enough	increase number of checkpoints, on page 11 – 8
undo segment extension	too much dynamic extension/ shrinking of rollback segments	prevent rollback segments from growing or shrinking by appropriately sizing them, on page 9 – 11
undo segment tx slot	not enough rollback segments	create the appropriate number of rollback segments, on page 10 – 3

Table 11 – 1 Causes of Events in V\$SESSION_WAIT

Determining the Cause of the Problem

Each performance problem has a unique cause, and you may need to query several dynamic performance views to find the cause of some problems. This manual groups related performance problems into chapters. The table in the previous section points you to the proper chapter for each type of problem. Performance problems tend to fall into one of these categories:

- CPU
- memory
- I/O
- contention for latches or other structures

The V\$SESSION_WAIT table helps you to categorize the performance problem and place it into one of these three categories. Also, there is additional information in columns P1, P2, and P3 for some events in V\$SESSION_WAIT. Once you have queried V\$SESSION_WAIT to get a general idea of the type of performance problem, refer to the appropriate chapter in this manual to diagnose the problem specifically.

Examine the values in P1, P2, and P3 of V\$SESSION_WAIT for the latch problems we discovered in the previous section. Use the following query to obtain additional information about the events that current sessions are waiting on.

```
SQL> SELECT sid, event, p1text, p1, p2text,
2      p2, p3text, p3, wait_time
3      FROM v$session_wait
4      WHERE event = 'latch free'
5      AND wait_time > 40000000;
```

SID	EVENT	P1TEXT	P1	P2TEXT	P2	P3TEXT	P3	WAIT_TIME
205	latch free	address	50367360	number	15	tries	1	4294967295
207	latch free	address	50367260	number	16	tries		4294967295
209	latch free	address	50367360	number	15	tries	1	4294967295
215	latch free	address	50367360	number	15	tries		4294967295
293	latch free	address	50367260	number	16	tries	1	4294967295
294	latch free	address	50367260	number	16	tries		4294967295

P2 tells us that three sessions each are currently waiting on latch numbers 15 and 16. We can find the names of those latches by querying V\$LATCH.

```
SQL> SELECT latch#, name
2      FROM v$latch
3      WHERE latch# IN (15,16);
LATCH# NAME
-----
15 redo allocation
16 redo copy
```

You can now see that the redo allocation and redo copy latches are the latches for which these sessions are waiting. The next step is to examine redo latch activity. Use the following query to determine the redo latch activity:

```
SQL> SELECT ln.name, gets, misses,
2          immediate_gets, immediate_misses
3      FROM v$latch l, v$latchname ln
4      WHERE ln.name IN('redo allocation', 'redo copy')
5          AND ln.latch# = l.latch#;
```

NAME	GETS	MISSES	IMMEDIATE_GETS	IMMEDIATE_MISSES
redo allocation	32624433	147985	0	0
redo copy	527	227	47615279	540

The following table illustrates the significant ratios for the statistics:

	redo allocation	redo copy
MISSES/GETS	4.5%	43%
IMMEDIATE_MISSES/ (IMMEDIATE_GETS + IMMEDIATE_MISSES)	0%	<0.01%

If either the ratio of MISSES to GETS or the ratio of IMMEDIATE_MISSES to the sum of IMMEDIATE_GETS and IMMEDIATE_MISSES is greater than 1%, there is a contention problem for the redo allocation or redo copy latches. In the example, the ratio of MISSES to GETS is greater than 1% for both the redo allocation and redo copy latches, so you must take corrective action.

The SQL Trace Facility

The SQL trace facility provides performance information on individual SQL statements. The SQL trace facility generates the following statistics for each statement:

- parse, execute, and fetch counts
- CPU and elapsed times
- physical reads and logical reads
- number of rows processed
- misses on the library cache

You can enable the SQL trace facility for a session or for an instance. When the SQL trace facility is enabled, performance statistics for all SQL statements executed in a user session or in an instance are placed into a trace file.

You can run the TKPROF program to format the contents of the trace file and place the output into a readable output file. As options, TKPROF can also

- determine the execution plans of SQL statements
- create a SQL script that stores the statistics in the database

Because running the SQL trace facility increases system overhead, you should enable it only when tuning your SQL statements, and disable it when you are finished.

Using the SQL Trace Facility

Follow these steps to use the SQL trace facility:

1. Set initialization parameters to prepare Oracle for using the SQL trace facility.
2. Enable the SQL trace facility for the desired session and run your application. This step produces a trace file containing statistics for the SQL statements issued by the application.
3. Run TKPROF to translate the trace file created in Step 2 into a readable output file. This step can also optionally create a SQL script that stores the statistics in the database.
4. Interpret the output file created in Step 3.
5. Optionally, run the SQL script produced in Step 3 to store the statistics in the database.

Each of these steps is discussed in the following sections.

**Setting Initialization
Parameters for the SQL
Trace Facility**

Before running your application with the SQL trace facility enabled, be sure these initialization parameter are set appropriately:

TIMED_STATISTICS	This parameter enables and disables the collection of timed statistics, such as CPU and elapsed times, by the SQL trace facility, as well as the collection of certain statistics in the dynamic performance tables. The default value of FALSE disables timing. A value of TRUE enables timing. Enabling timing causes extra timing calls for low-level operations.
MAX_DUMP_FILE_SIZE	This parameter specifies the maximum size of trace files in operating system blocks. The default is 500. If you find that your trace output is truncated, increase the value of this parameter before generating another trace file.
USER_DUMP_DEST	This parameter specifies the destination for the trace file. The destination must be fully specified according to the conventions of your operating system. The default value for this parameter is the default destination for system dumps on your operating system.

**Enabling the SQL
Trace Facility**

You can enable the SQL trace facility for either

- an individual session
- the instance

**Enabling Tracing for a
Session**

To enable the SQL trace facility for your session, issue this SQL statement:

```
ALTER SESSION
  SET SQL_TRACE = TRUE;
```

To disable the SQL trace facility for your session, issue this SQL statement:

```
ALTER SESSION
  SET SQL_TRACE = FALSE;
```

You can also enable the SQL trace facility for your session by using the DBMS_SESSION.SET_SQL_TRACE procedure.

You may need to modify your application to contain the ALTER SESSION command. For example, to issue the ALTER SESSION command in Oracle Forms, invoke Oracle Forms using the -s option, or invoke Oracle Forms (Design) using the statistics option. For more information on Oracle Forms, see the *Oracle Forms Reference* manual.

The SQL trace facility is also automatically disabled for your session when your application disconnects from Oracle.

Calling the DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION procedure enables the SQL trace facility for a session other than your current session. This procedure allows you to gather statistics for a different user's session. This can be useful for database administrators who are not located near their users or who do not have access to the application code to set SQL trace from within an application. This procedure requires the session id and serial number of the user session in which you wish to enable SQL trace.

You can obtain the session id and serial number from the V\$SESSION view. The following is an example of a Server Manager line mode session that obtains the session id and serial number for the user JFRAZZIN and then enables SQL trace for that user's session:

```
SVRMGR> SELECT sid, serial#, osuser
2>      FROM v$session
3>      WHERE osuser = 'jfrazzin';
```

SID	SERIAL#	OSUSER
8	12	jfrazzin

1 row selected.

```
SVRMGR> EXECUTE dbms_system.set_sql_trace_in_session(8,12,TRUE);
Statement processed.
```

Enabling the SQL Trace Facility for an Instance

To enable the SQL trace facility for your instance, set the value of the initialization parameter SQL_TRACE to TRUE. This value causes statistics to be collected for all sessions.

Once the SQL trace facility has been enabled for the instance, it may be disabled for an individual session with this SQL statement:

```
ALTER SESSION
SET SQL_TRACE = FALSE;
```

Generating Trace Files

When the SQL trace facility is enabled for a session, Oracle generates a trace file containing statistics for traced SQL statements for that session. When the SQL trace facility is enabled for an instance, Oracle creates a separate trace file for each process.

Because Oracle writes these trace files to the user dump destination, be sure you know how to distinguish them by name.

If your operating system retains multiple versions of files, be sure your version limit is high enough to accommodate the number of trace files you expect the SQL trace facility to generate.

The generated trace files may be owned by an operating system user other than yourself. This user must make the trace files available to you before you can use TKPROF to format them.

Once the SQL trace facility has generated a number of trace files, you can either:

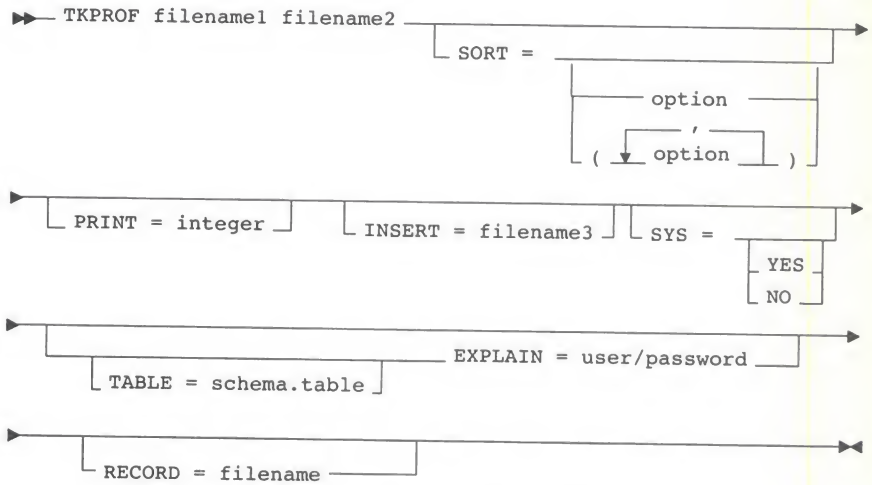
- run TKPROF on each individual trace file, producing a number of formatted output files, one for each session
- append the trace files together and then run TKPROF on the result to produce a formatted output file for the entire instance.

Note: Trace files generated immediately after instance startup contain data that reflects the activity of the startup process. In particular, they reflect a disproportionate amount of I/O activity as caches in the System Global Area (SGA) are filled. For the purposes of tuning, ignore such trace files.

Running TKPROF

TKPROF accepts as input a trace file produced by the SQL trace facility and produces a formatted output file. TKPROF can also be used to generate execution plans. Invoke TKPROF using this syntax:

TKPROF command ::=



If you invoke TKPROF with no arguments, online help is displayed.

Use the following arguments with TKPROF:

<i>filename1</i>	Specifies the input file, a trace file containing statistics produced by the SQL trace facility. This file can be either a trace file produced for a single session or a file produced by appending together individual trace files from multiple sessions.
<i>filename2</i>	Specifies the file to which TKPROF writes its formatted output.
EXPLAIN	Determines the execution plan for each SQL statement in the trace file and writes these execution plans to the output file. TKPROF determines execution plans by issuing the EXPLAIN PLAN command after connecting to Oracle with the <i>user</i> and <i>password</i> specified in this parameter. The specified <i>user</i> must have CREATE SESSION system privileges.
TABLE	Specifies the <i>schema</i> and name of the <i>table</i> into which TKPROF temporarily places execution plans before writing them to the output file. If the specified table already exists, TKPROF deletes its rows, uses it for the EXPLAIN PLAN command, and then deletes its rows. If this table does not exist, TKPROF creates it, uses it, and then drops it.

The specified *user* must be able to issue INSERT, SELECT, and DELETE statements against the table. If the table does not already exist, the user must also be able to issue CREATE TABLE and DROP TABLE statements.

For the privileges to issue these statements, see the *Oracle7 Server SQL Reference*.

This option allows multiple individuals to run TKPROF concurrently with the same *user* in the EXPLAIN value. These individuals can specify different TABLE values and avoid destructively interfering with each other's processing on the temporary plan table.

If you use the EXPLAIN parameter without the TABLE parameter, TKPROF uses the table PROF\$PLAN_TABLE in the schema of the *user* specified by the EXPLAIN parameter. If you use the TABLE parameter without the EXPLAIN parameter, TKPROF ignores the TABLE parameter.

INSERT
Creates a SQL script that stores the trace file statistics in the database. TKPROF creates this script with the name *filename3*. This script creates a table and inserts a row of statistics for each traced SQL statement into the table.

SYS
Enables and disables the listing of SQL statements issued by the user SYS, or recursive SQL statements into the output file. The default value of YES causes TKPROF to list these statements. The value of NO causes TKPROF to omit them.

Note that this parameter does not affect the optional SQL script. The SQL script always inserts statistics for all traced SQL statements, including recursive SQL statements.

SORT
Sorts the traced SQL statements in descending order of the specified sort option before listing them into the output file. If more than one option is specified, the output is sorted in descending order by the sum of the values specified in the sort options. If you omit this parameter, TKPROF lists statements into the output file in ascending order of when each was first issued.

The sort options are

PRSCNT	number of times parsed
PRSCPU	CPU time spent parsing
PRSELA	elapsed time spent parsing
PRSDSK	number of physical reads from disk during parse
PRSQRY	number of consistent mode block reads during parse
PRSCU	number of current mode block reads during parse
PRSMIS	number of library cache misses during parse
EXECNT	number of executes
EXECPU	CPU time spent executing
EXEELA	elapsed time spent executing
EXEDSK	number of physical reads from disk during execute
EXEQRY	number of consistent mode block reads during execute
EXECU	number of current mode block reads during execute
EXEROW	number of rows processed during execute
EXEMIS	number of library cache misses during execute
FCHCNT	number of fetches

	FCHCPU	CPU time spent fetching
	FCHELA	elapsed time spent fetching
	FCHDSK	number of physical reads from disk during fetch
	FCHQRY	number of consistent mode block reads during fetch
	FCHCU	number of current mode block reads during fetch
	FCHROW	number of rows fetched
PRINT	Lists only the first <i>integer</i> sorted SQL statements into the output file. If you omit this parameter, TKPROF lists all traced SQL statements. Note that this parameter does not affect the optional SQL script. The SQL script always inserts statistics for all traced SQL statements.	
RECORD	Creates a SQL script with the specified filename with all of the non-recursive SQL in the trace file. This can be used to replay the user events from the trace file.	

Example This example runs TKPROF, accepts a trace file named DLSUN12_JOHN_FG_SVRMGR_007.TRC, and writes a formatted output file named OUTPUTA.PRF:

```
TKPROF DLSUN12_JOHN_FG_SVRMGR_007.TRC OUTPUTA.PRF
EXPLAIN=SCOTT/TIGER TABLE=SCOTT.TEMP_PLAN_TABLE_A
INSERT=STOREA.SQL SYS=NO SORT=(EXECPU,FCHCPU) PRINT=10
```

This example is likely to be longer than a single line on your terminal screen and you may have to use continuation characters, depending on your operating system.

Note the other parameters in this example:

- The EXPLAIN value causes TKPROF to connect as the user SCOTT and use the EXPLAIN PLAN command to generate the execution plan for each traced SQL statement.
- The TABLE value causes TKPROF to use the table TEMP_PLAN_TABLE_A in the schema SCOTT as a temporary plan table.
- The INSERT value causes TKPROF to generate a SQL script named STOREA.SQL that stores statistics for all traced SQL statements in the database.
- The SYS parameter with the value of NO causes TKPROF to omit recursive SQL statements from the output file.
- The SORT value causes TKPROF to sort the SQL statements in order of the sum of the CPU time spent executing and the CPU time spent fetching rows before writing them to the output file.
- The PRINT value of 10 causes TKPROF to write statistics for the first 10 sorted SQL statements to the output file.

Interpreting TKPROF Output

The following listing shows TKPROF output for one SQL statement as it appears in the output file:

SELECT * FROM emp, dept WHERE emp.deptno = dept.deptno;

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.16	0.29	3	13	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	1	0.03	0.26	2	2	4	14

Misses in library cache during parse: 1
Parsing user id: 8

Rows	Execution Plan
14	MERGE JOIN
4	SORT JOIN
4	TABLE ACCESS (FULL) OF 'DEPT'
14	SORT JOIN
14	TABLE ACCESS (FULL) OF 'EMP'

For this statement, TKPROF output has these parts:

- the text of the SQL statement
- the SQL trace statistics in tabular form
- the number of library cache misses for the parsing and execution of the statement
- the user initially parsing the statement
- the execution plan generated by EXPLAIN PLAN

Note: TKPROF cannot tell the TYPE of the bind variables simply by looking at the text of the SQL statement. It assumes that TYPE is CHARACTER; if this is not the case, you should put appropriate type conversions in the SQL statement.

SQL Trace Facility
Statistics

TKPROF lists the statistics for a SQL statement returned by the SQL trace facility in rows and columns. Each row corresponds to one of three steps of SQL statement processing:

Parse	This step translates the SQL statement into an execution plan. This step includes checks for proper security authorization and checks for the existence of tables, columns, and other referenced objects.
Execute	This step is the actual execution of the statement by Oracle. For INSERT, UPDATE, and DELETE statements, this step modifies the data. For SELECT statements, the step identifies the selected rows.
Fetch	This step retrieves rows returned by a query. Fetches are only performed for SELECT statements.

The step for which each row contains statistics is identified by the value of the *call* column.

The other columns of the SQL trace facility output are combined statistics for all parses, all executes, and all fetches of a statement:

<i>count</i>	Number of times a statement was parsed, executed, or fetched.
<i>cpu</i>	Total CPU time in seconds for all parse, execute, or fetch calls for the statement.
<i>elapsed</i>	Total elapsed time in seconds for all parse, execute, or fetch calls for the statement.

<i>disk</i>	Total number of data blocks physically read from the datafiles on disk for all parse, execute, or fetch calls.
<i>query</i>	Total number of buffers retrieved in consistent mode for all parse, execute, or fetch calls. Buffers are usually retrieved in consistent mode for queries.
<i>current</i>	Total number of buffers retrieved in current mode. Buffers are often retrieved in current mode for INSERT, UPDATE, and DELETE statements. The sum of <i>query</i> and <i>current</i> is the total number of buffers accessed.
<i>rows</i>	Total number of rows processed by the SQL statement. This total does not include rows processed by subqueries of the SQL statement. For SELECT statements, the number of rows returned appears for the fetch step. For UPDATE, DELETE, and INSERT statements, the number of rows processed appears for the execute step.

Resolution of Statistics Since timing statistics have a resolution of one hundredth of a second, any operation on a cursor that takes a hundredth of a second or less may not be timed accurately. Keep this in mind when interpreting statistics. In particular, be careful when interpreting the results from simple queries that execute very quickly.

Recursive Calls Sometimes to execute a SQL statement issued by a user, Oracle must issue additional statements. Such statements are called *recursive calls* or *recursive SQL statements*. For example, if you insert a row into a table that does not have enough space to hold that row, Oracle makes recursive calls to allocate the space dynamically. Recursive calls are also generated when data dictionary information is not available in the data dictionary cache and must be retrieved from disk.

If recursive calls occur while the SQL trace facility is enabled, TKPROF produces statistics for the recursive SQL statements and clearly marks them as recursive SQL statements in the output file. Note that the statistics for a recursive SQL statement are included in the listing for that statement, not in the listing for the SQL statement that caused the recursive call. So when you are calculating the total resources required to process a SQL statement, you should consider the statistics for that

	statement as well as those for recursive calls caused by that statement. Note that setting the TKPROF command line parameter to NO suppresses the listing of recursive calls in the output file.
Library Cache Misses	TKPROF also lists the number of library cache misses resulting from parse and execute steps for each SQL statement. These statistics appear on separate lines following the tabular statistics. If the statement resulted in no library cache misses, TKPROF does not list the statistic. In the example, the statement resulted in one library cache miss for the parse step and no misses for the execute step.
User Issuing the SQL Statement	TKPROF also lists the user ID of the user issuing each SQL statement. If the TKPROF input file contained statistics from multiple users and the statement was issued by more than one user, TKPROF lists the ID of the last user to parse the statement. The user ID of all database users appears in the data dictionary in the column ALL_USERS.USER_ID.
Execution Plan	If you specify the EXPLAIN parameter on the TKPROF command line, TKPROF uses the EXPLAIN PLAN command to generate the execution plan of each SQL statement traced. For more information on interpreting execution plans, see the section “Example of EXPLAIN PLAN Output” on page A – 27. TKPROF also displays the number of rows processed by each step of the execution plan.
Storing SQL Trace Facility Statistics	<p>You may want to keep a history of the statistics generated by the SQL trace facility for your application and compare them over time. TKPROF can generate a SQL script that creates a table and inserts rows of statistics into it. This script contains</p> <ul style="list-style-type: none"> • a CREATE TABLE statement that creates an output table named TKPROF_TABLE • INSERT statements that add rows of statistics, one for each traced SQL statement, to the TKPROF_TABLE <p>After running TKPROF, you can run this script to store the statistics in the database.</p>
Generating the TKPROF Output SQL Script	When you run TKPROF, use the INSERT parameter to specify the name of the generated SQL script. If you omit this parameter, TKPROF does not generate a script.

Editing the TKPROF
Output SQL Script

After TKPROF has created the SQL script, you may want to edit the script before running it:

- If you have already created an output table for previously collected statistics and you want to add new statistics to the existing table, remove the CREATE TABLE statement from the script. The script will then insert the new rows into the existing table.
- If you have created multiple output tables, perhaps to store statistics from different databases in different tables, edit the CREATE TABLE and INSERT statements to change the name of the output table.

Querying the Output
Table

The following CREATE TABLE statement creates the TKPROF_TABLE:

```
CREATE TABLE tkprof_table
(date_of_insert    DATE,
cursor_num        NUMBER,
depth             NUMBER,
user_id           NUMBER,
parse_cnt         NUMBER,
parse_cpu         NUMBER,
parse_elap        NUMBER,
parse_disk        NUMBER,
parse_query       NUMBER,
parse_current     NUMBER,
parse_miss        NUMBER,
exe_count         NUMBER,
exe_cpu           NUMBER,
exe_elap          NUMBER,
exe_disk          NUMBER,
exe_query         NUMBER,
exe_current       NUMBER,
exe_miss          NUMBER,
exe_rows          NUMBER,
fetch_count       NUMBER,
fetch_cpu         NUMBER,
fetch_elap        NUMBER,
fetch_disk        NUMBER,
fetch_query       NUMBER,
fetch_current     NUMBER,
fetch_rows        NUMBER,
clock_ticks       NUMBER,
sql_statement     LONG)
```

These columns help you identify a row of statistics by these columns:

SQL_STATEMENT	The column value is the SQL statement for which the SQL trace facility collected the row of statistics. Note that because this column has datatype LONG, you cannot use it in expressions or WHERE clause conditions.
DATE_OF_INSERT	The column value is the date and time when the row was inserted into the table. Note that this value is not exactly the same as the time the statistics were collected by the SQL trace facility. Most output table columns correspond directly to the statistics that appear in the formatted output file. For example the PARSE_CNT column value corresponds to the <i>count</i> statistic for the parse step in the output file.
DEPTH	This column value indicates the level of recursion at which the SQL statement was issued. For example, a value of 1 indicates that a user issued the statement. A value of 2 indicates Oracle generated the statement as a recursive call to process a statement with a value of 1 (a statement issued by a user). A value of <i>n</i> indicates Oracle generated the statement as a recursive call to process a statement with a value of <i>n</i> -1.
USER_ID	This column value identifies the user issuing the statement. This value also appears in the formatted output file.
CURSOR_NUM	This column value is used by Oracle to keep track of the cursor to which each SQL statement was assigned. Note that the output table does not store the statement's execution plan.

Example This query returns the statistics from the output table. These statistics correspond to the formatted output shown in the section "Interpreting TKPROF Output" on page A - 15.

SELECT * FROM tkprof_table;

DATE_OF_INSERT	CURSOR_NUM	DEPTH	USER_ID	PARSE_CNT	PARSE_CPU	PARSE_ELAP
27-OCT-1993	1	0	8	1	16	29
PARSE_DISK	PARSE_QUERY	PARSE_CURRENT	PARSE_MISS	EXE_COUNT	EXE_CPU	
3	13	0	1	1	0	
EXE_ELAP	EXE_DISK	EXE_QUERY	EXE_CURRENT	EXE_MISS	EXE_ROWS	FETCH_COUNT
0	0	0	0	0	0	1
FETCH_CPU	FETCH_ELAP	FETCH_DISK	FETCH_QUERY	FETCH_CURRENT	FETCH_ROWS	
3	26	2	2	4	14	

SQL_STATEMENT

SELECT * FROM EMP, DEPT WHERE EMP.DEPTNO = DEPT.DEPTNO

The EXPLAIN PLAN Command

The EXPLAIN PLAN command displays the execution plan chosen by the Oracle optimizer for SELECT, UPDATE, INSERT, and DELETE statements. A statement's execution plan is the sequence of operations that Oracle performs to execute the statement. By examining the execution plan, you can see exactly how Oracle executes your SQL statement. This information can help you determine whether the SQL statement you have written takes advantage of the indexes available. For the syntax of the EXPLAIN PLAN command, see the *Oracle7 Server SQL Reference*.

Creating the Output Table

Before you can issue an EXPLAIN PLAN statement, there must exist a table to hold its output. You can create an output table through either of these means:

- You can run the SQL script UTLXPLAN.SQL to create a sample output table called PLAN_TABLE in your schema. The exact name and location of this script may vary depending on your operating system. PLAN_TABLE is the default table into which the EXPLAIN PLAN statement inserts rows describing execution plans.
- You can issue a CREATE TABLE statement to create an output table with any name you choose. In this case, you can issue an EXPLAIN PLAN statement and direct its output to this table.

Any table used to store the output of the EXPLAIN PLAN command must have the same column names and datatypes as this PLAN_TABLE:

```
CREATE TABLE plan_table
(statement_id      VARCHAR2(30),
timestamp         DATE,
remarks           VARCHAR2(80),
operation         VARCHAR2(30),
options           VARCHAR2(30),
object_node       VARCHAR2(128),
object_owner      VARCHAR2(30),
object_name       VARCHAR2(30),
object_instance   NUMERIC,
object_type       VARCHAR2(30),
optimizer         VARCHAR2(255),
search_columns    NUMERIC,
id               NUMERIC,
parent_id        NUMERIC,
position         NUMERIC,
other            LONG);
```

Output Table Columns The PLAN_TABLE used by the EXPLAIN PLAN command contains the following columns:

STATEMENT_ID	The value of the option STATEMENT_ID parameter specified in the EXPLAIN PLAN statement.
TIMESTAMP	The date and time when the EXPLAIN PLAN statement was issued.
REMARKS	Any comment (of up to 80 bytes) you wish to associate with each step of the explained plan. If you need to add or change a remark on any row of the PLAN_TABLE, use the UPDATE statement to modify the rows of the PLAN_TABLE.
OPERATION	<p>The name of the internal operation performed in this step.</p> <p>In the first row generated for a statement, the column contains one of these values, depending on the type of the statement:</p> <p>'DELETE STATEMENT' 'INSERT STATEMENT' 'SELECT STATEMENT' 'UPDATE STATEMENT'</p>
OPTIONS	A variation on the operation described in the OPERATION column.
OBJECT_NODE	The name of the database link used to reference the object (a table name or view name). For local queries using the parallel query option, this column describes the order in which output from operations is consumed.
OBJECT_OWNER	The name of the user that owns the schema containing the table or index.
OBJECT_NAME	The name of the table or index.
OBJECT_INSTANCE	A number corresponding to the ordinal position of the object as it appears in the original statement. The numbering proceeds from left to right, outer to inner with respect to the original statement text. Note that view expansion will result in unpredictable numbers.

OBJECT_TYPE	A modifier that provides descriptive information about the object; for example, NON-UNIQUE for indexes.
OPTIMIZER	The current mode of the optimizer.
SEARCH_COLUMNS	Not currently used.
ID	A number assigned to each step in the execution plan.
PARENT_ID	The ID of the next execution step that operates on the output of the ID step.
POSITION	The order of processing for steps that all have the same PARENT_ID.
OTHER	Other information that is specific to the execution step that a user may find useful.
OTHER_TAG	Describes the contents of the OTHER column. See NO TAG for more information on the possible values for this column.
COST	The cost of the operation as estimated by the optimizer's cost-based approach. For statements that use the rule-based approach, this column is null. Cost is not determined for table access operations. The value of this column does not have any particular unit of measurement, it is merely a weighted value used to compare costs of execution plans.
CARDINALITY	The cost-based approach's estimate of the number of rows accessed by the operation.
BYTES	The cost-based approach's estimate of the number of bytes accessed by the operation.

Table A – 1 describes the possible values of the OTHER_TAG column.

Column Contents	Description
SERIAL	The statement is a locally executed serial query plan.
SERIAL_FROM_REMOTE	The statement is executed at a remote site.
PARALLEL_COMBINED_WITH_PARENT	The parent of this operation is an operator that performs the parent and child operation together.
PARALLEL_COMBINED_WITH_CHILD	The child of this operation is an operator that performs the parent and child operation together.
PARALLEL_TO_SERIAL	The SQL in the OTHER column is the top level of the parallel execution plan.
PARALLEL_TO_PARALLEL	This operation consumes data from a parallel source and outputs its data in parallel.
PARALLEL_FROM_SERIAL	This operation consumes data from a serial operation and outputs its data in parallel.

Table A – 1 Possible Values of the OTHER_TAG column of the PLAN Table.

Table A – 2 lists each combination of OPERATION and OPTION values produced by the EXPLAIN PLAN command and its meaning within an execution plan.

OPERATION	OPTION	Description
AND-EQUAL		An operation that accepts multiple sets of ROWIDS and returns the intersection of the sets, eliminating duplicates. This operation is used for the single-column indexes access path.
CONNECT BY		A retrieval of rows in a hierarchical order for a query containing a CONNECT BY clause.
CONCATENATION		An operation that accepts multiple sets of rows and returns the union—all of the sets.
COUNT		An operation that counts the number of rows selected from a table.
	STOPKEY	A count operation where the number of rows returned is limited by the ROWNUM expression in the WHERE clause.
FILTER		An operation that accepts a set of rows, eliminates some of them, and returns the rest.
FIRST ROW		A retrieval on only the first row selected by a query.

Table A – 2 OPERATION and OPTION Values Produced by the EXPLAIN PLAN Command.

OPERATION	OPTION	Description
FOR UPDATE		An operation that retrieves and locks the rows selected by a query containing a FOR UPDATE clause.
INDEX*	UNIQUE SCAN	A retrieval of a single ROWID from an index.
	RANGE SCAN	A retrieval of one or more ROWIDs from an index. Indexed values are scanned in ascending order.
	RANGE SCAN DESCENDING	A retrieval of one or more ROWIDs from an index. Indexed values are scanned in descending order.
INTERSECTION		An operation that accepts two sets of rows and returns the intersection of the sets, eliminating duplicates.
MERGE JOIN+		An operation that accepts two sets of rows, each sorted by a specific value, combines each row from one set with the matching rows from the other, and returns the result.
	OUTER	A merge join operation to perform an outer join statement.
CONNECT BY		A retrieval of rows in hierarchical order for a query containing a CONNECT BY clause.
MINUS		An operation that accepts two sets of rows and returns rows that appear in the first set but not in the second, eliminating duplicates.
NESTED LOOPS+		An operation that accepts two sets of rows, an outer set and an inner set. Oracle compares each row of the outer set with each row of the inner set and returns those rows that satisfy a condition.
NESTED LOOPS+	OUTER	A nested loops operation to perform an outer join statement.
PROJECTION		An internal operation.
REMOTE		A retrieval of data from a remote database.
SEQUENCE		An operation involving accessing values of a sequence.
SORT	AGGREGATE	A retrieval of a single row that is the result of applying a group function to a group of selected rows.

Table A – 2 OPERATION and OPTION Values Produced by the EXPLAIN PLAN Command.

OPERATION	OPTION	Description
	UNIQUE	An operation that sorts a set of rows to eliminateduplicates.
	GROUP BY	An operation that sorts a set of rows into groups for a query with a GROUP BY clause.
	JOIN	An operation that sorts a set of rows before a merge-join operation.
	ORDER BY	An operation that sorts a set of rows for a query with an ORDER BY clause.
TABLE ACCESS*	FULL	A retrieval of all rows from a table.
	CLUSTER	A retrieval of rows from a table based on a value of the key of an indexed cluster.
	HASH	A retrieval of rows from a table based on a value of the key of hash cluster.
	BY ROWID	A retrieval of a row from a table based on its ROWID.
UNION		An operation that accepts two sets of rows and returns the union of the sets, eliminatingduplicates.
VIEW		An operation that performs a view's query and then returns the resulting rows to another operation.

Table A – 2 OPERATION and OPTION Values Produced by the EXPLAIN PLAN Command.

* These operations are access methods.
 + These operations are join operations.

Both access methods and join operations are discussed in Chapter 5, “The Optimizer”.

Example of EXPLAIN PLAN Output

The following example shows a SQL statement and its corresponding execution plan generated by using the EXPLAIN PLAN command.

This query retrieves names and related information for employees whose salary is not within any range of the SALGRADE table:

```
SELECT ename, job, sal, dname
FROM emp, dept
WHERE emp.deptno = dept.deptno
AND NOT EXISTS
  (SELECT *
   FROM salgrade
   WHERE emp.sal BETWEEN losal AND hisal);
```

This EXPLAIN PLAN statement generates an execution plan and places the output in PLAN_TABLE:


```
EXPLAIN PLAN
SET STATEMENT_ID = 'Emp_Sal'
FOR SELECT ename, job, sal, dname
FROM emp, dept
WHERE emp.deptno = dept.deptno
AND NOT EXISTS
(SELECT *
FROM salgrade
WHERE emp.sal BETWEEN losal AND hisal);
```

Table Format of
EXPLAIN PLAN Output

This SELECT statement generates the following output:

```
SELECT operation, options, object_name, id, parent_id, position
FROM plan_table
WHERE statement_id = 'Emp_Sal'
ORDER BY id;
```

OPERATION	OPTIONS	OBJECT_NAME	ID	PARENT_ID	POSITION
SELECT STATEMENT			0		5
FILTER			1	0	0
NESTED LOOPS			2	1	1
TABLE ACCESS	FULL	EMP	3	2	1
TABLE ACCESS	FULL	DEPT	4	2	2
TABLE ACCESS	FULL	SALGRADE	5	1	3

The ORDER BY clause returns the steps of the execution plan sequentially by ID value. However, Oracle does not perform the steps in this order. Since PARENT_ID receives information from ID, observe that more than one ID step fed PARENT_ID. For example, step 2, a merge join, and step 7, a table access, both fed step 1. A nested, visual representation of the processing sequence is shown in the next section.

The value of the POSITION column for the first row of output indicates that the optimizer estimates the cost of executing the statement with this execution plan to be 5.

Nested Format of
EXPLAIN PLAN Output

This type of SELECT statement generates a nested representation of the output that more closely depicts the order of steps undergone in processing the SQL statement. The order resembles a “tree structure”, illustrated in Figure A – 1.

```
SELECT LPAD(' ',2*(LEVEL-1))||operation||' '||options
||' '||object_name
||' '||DECODE(id, 0, 'Cost = '||position) "Query Plan"
FROM plan_table
START WITH id = 0 AND statement_id = 'Emp_Sal'
CONNECT BY PRIOR id = parent_id AND statement_id = 'Emp_Sal';
```

Query Plan

```

SELECT STATEMENT   Cost = 5
  FILTER
    NESTED LOOPS
      TABLE ACCESS FULL EMP
      TABLE ACCESS FULL DEPT
      TABLE ACCESS FULL SALGRADE

```

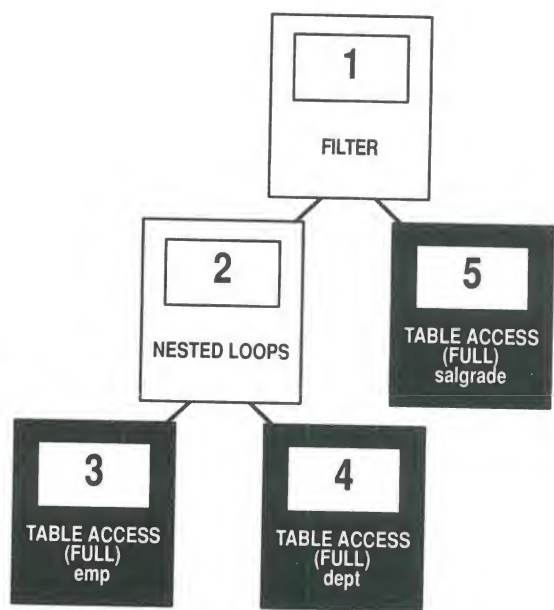


Figure A – 1 Tree Structure of an Execution Plan

The tree structure illustrates how operations that occur during the execution of a SQL statement feed one another. Each step in the execution plan is assigned a number (representing the ID column of the PLAN_TABLE) and is depicted by a “node”. The result of each node’s operation passes to its parent node, which uses it as input.

Oracle Trace

Oracle Trace collects performance data for any application—most notably, transaction processing and database applications. It provides for the gathering and reporting of event-based performance data from layered products and application programs that contain calls to Oracle Trace routines. Oracle Trace is designed to operate with minimal performance impact on the system and can be used in both development and production environments.

Oracle Trace can be used to assist in pinpointing the reasons for an application's poor performance. General reasons for poor performance can be any of the following:

- data access contention
- poorly or incorrectly designed databases
- not enough servers to handle user requests
- inefficient database queries
- actual use of the application differs from the intended use
- inadequate hardware resources

Finding specific causes for these general problems requires data about the application's resource use and response time. Oracle Trace collects a variety of such data from all layers of an application—the user interface, the processing engine, and the database.

Instrumentation of Oracle For Oracle Trace

The Oracle Server has been instrumented for use with Oracle Trace. To gather statistics, Oracle Trace requires a facility definition file. For more information about instrumentation and facility definition files, see the *Oracle Trace Instrumentation Guide*.

Enabling and Disabling Oracle Trace

Oracle Trace can be enabled and configured on an instance-wide or per-session basis. To enable and configure Oracle Trace for the entire instance, you must set `ORACLE_TRACE_ENABLED` to `TRUE`, and set other `ORACLE_TRACE_*` initialization parameters for the destination of the collection, the size of the collection, and so on. For information about these parameters, see the *Oracle7 Server Reference*.

To enable and configure Oracle Trace for an individual session, use the `DBMS_ORACLE_TRACE_USER.SET_ORACLE_TRACE` procedure.

B

Statistic Descriptions

This appendix briefly describes some of the statistics stored in the V\$SESSION_WAIT and V\$SYSSTAT dynamic performance table. These statistics are useful in identifying and correcting performance problems.

Statistic Descriptions

buffer busy waits	<p>This statistic is stored in V\$SESSION_WAIT. In single-instance mode, concurrent I/O on a database block causes this statistic to increment. High buffer busy waits usually indicates a predominately I/O bound application.</p> <p>In an Oracle Parallel Server, it is common to wait for the buffer as the Distributed File System (DFS) lock gets escalated to Exclusive Mode. High buffer busy waits in an Oracle Parallel Server usually indicates competition between nodes for database blocks.</p> <p>P1 in the V\$SESSION_WAIT table for this statistic is the database file number. P2 is the database block offset into that file. P3 uniquely identifies where in the RDBMS the event was triggered.</p>
consistent changes	<p>This statistic is stored in V\$SYSSTAT. This statistic indicates the number of times a database block has rollback entries applied to perform a consistent read on the block.</p> <p>Workloads that produce a great deal of consistent changes can consume a great deal of resources.</p>
consistent gets	<p>This statistic is stored in V\$SYSSTAT. This statistic indicates the number of times a consistent read was requested for a block. See also "consistent changes".</p>
db block changes	<p>This statistic is stored in V\$SYSSTAT. Closely related to Consistent changes, this statistics counts the total number of changes that were made to all blocks in the SGA that were part of an update or delete operation. These are changes that are generating redo log entries and hence will be permanent changes to the database if the transaction is committed.</p> <p>This statistic is a rough indication of total database work. This statistic indicates (possibly on a per transaction level) the rate at which buffers are being dirtied.</p>
db block gets	<p>This statistic is stored in V\$SYSSTAT. This statistic tracks the number of blocks gotten in current mode.</p>

free buffer waits	<p>This statistic is stored in V\$SYSSTAT. This statistic stores the number of times a free buffer was requested in the SGA, but none were available. Free buffers are buffers that are not currently being used by other database users.</p> <p>If the SGA is full of dirty buffers and DBWR can't write them to disk, then free buffer waits will increase. Update-intensive applications that use small indexes or hash clusters may run the risk of having an entire SGA full of dirty buffers that DBWR cannot keep up with.</p>
parse count	<p>This statistic is stored in V\$SYSSTAT. Independent of any cache benefits that may occur in the library cache, this statistic counts the number of times the user called 'parse' function from either OCI or the Oracle Precompilers.</p> <p>This statistic is used mostly as a denominator in conjunction with the V\$LIBRARY_CACHE table to determine actual hit ratios in the shared SQL area.</p>
physical reads	<p>This statistic is stored in V\$SYSSTAT. This statistic stores the number of I/O requests to the operating system to retrieve a database block from the disk subsystem. This is a buffer cache miss.</p> <p>Logical reads is consistent gets + database block gets. Logical reads and physical reads is used to calculate the buffer cache hit ratio.</p>
physical writes	<p>This statistic is stored in V\$SYSSTAT. This statistic stores the number of I/O requests to the operating system to write a database block to the disk subsystem. The bulk of the writes are performed either by DBWR or LGWR.</p>
recursive calls	<p>This statistic is stored in V\$SYSSTAT. Oracle maintains tables used for internal processing. When Oracle needs to make a change to these tables, it internally generates a SQL statement. These internal SQL statements generate recursive calls.</p>
redo entries	<p>This statistic is stored in V\$SYSSTAT. This statistic increments each time redo entries are copied into the redo log buffer.</p>

redo log space requests	<p>This statistic is stored in V\$SYSSTAT. The active log file has filed up and Oracle is waiting for disk space to be allocated for the redo log entries. Space is created by performing a log switch.</p> <p>Small Log files in relation to the size of the SGA or the commit rate of the workload can cause problems. When the log switch occurs, Oracle must ensure that all committed dirty buffers are written to disk before switching to a new log file. If you have a large SGA full of dirty buffers and small redo log files, a log switch must wait for DBWR to write dirty buffers to disk before continuing.</p> <p>Also examine the 'log file space/switch' wait event in V\$SESSION_WAIT.</p>
redo sync writes	<p>This statistic is stored in V\$SYSSTAT. Normally, redo that is generated and copied into the log buffer need not be flushed out to disk immediately. The log buffer is a circular buffer that LGWR periodically flushes. Redo sync writes increments when changes being applied must be written out to disk due to a commit.</p>
sorts (disk)	<p>This statistic is stored in V\$SYSSTAT. If the number of disk writes is non-zero for a given sort operation, then this statistic is incremented.</p> <p>Sorts that require I/O to disk are quite resource intensive. Try increasing the initialization parameter SORT_AREA_SIZE.</p>
sorts (memory)	<p>This statistic is stored in V\$SYSSTAT. If the number of disk writes is zero, then the sort was performed completely in memory and this statistic is incremented.</p> <p>This is more an indication of sorting activity in the application workload. You can't do much better than memory sorts, except maybe no sorts at all. Sorting is usually caused by selection criteria specifications within table join SQL operations.</p>
table fetch rowid	<p>This statistic is stored in V\$SYSSTAT. When rows are fetched using a rowid (usually recovered from an index), each row returned increments this counter.</p> <p>This statistic is an indication of row fetch operations being performed with the aid of an index. Because doing table scans usually either indicates non-optimal queries or tables without indexes, this statistic should increase as the above issues have been addressed in the application.</p>

table fetch continued row

This statistic is stored in V\$SYSSTAT. When a row that spans more than one block is encountered during a fetch, this statistic is incremented.

Retrieving rows that span more than one block increases the logical I/O by factor that corresponds to the number of blocks than need to be accessed. Exporting and re-importing may eliminate this problem. Taking a closer look at the STORAGE parameters PCT_FREE and PCT_USED. This problem cannot be fixed if rows are larger than database blocks (for example, if the LONG datatype is used and the rows are extremely large).

table scan blocks

This statistic is stored in V\$SYSSTAT. During scanning operations, each row is retrieved sequentially by Oracle and every each block encountered during the scan increments this statistic.

Informs you the number of database blocks you had to get from the buffer cache for the purpose of scanning. Compare this to consistent gets to get a feeling for how much of the consistent read activity can be attributed to scanning.

table scan rows

This statistic is stored in V\$SYSSTAT. This statistic is collected during a scan operation, but instead of counting the number of database blocks, it counts the rows being processed.

table scans (long tables)

This statistic is stored in V\$SYSSTAT. Long (or conversely short) tables can be defined as tables that don't meet the short table criteria as described in "table scans (short)".

table scans (short tables)

This statistic is stored in V\$SYSSTAT. Long (or conversely short) tables can be defined by optimizer hints coming down into the row source access layer of Oracle. The table must be below the initialization parameter SMALL_TABLE_THRESHOLD. In Parallel Server environments that support parallel query servers across instances, it can be determined if the cache partition will fit in the current SGA. If it does, then the table is considered small for scanning purposes.

Users with a lot of small tables that qualify under the above conditions benefit greatly by having these tables cached in the SGA. For parallel query operations, the merge phase of the query will improve as the pieces to be merged have been cached.

user calls

This statistic is stored in V\$SYSSTAT. Oracle allocates resources (Call State Objects) to keep track of relevant user call data structures every time you login, parse, or execute.

When determining activity, the ratio of user calls to RPI calls, give you an indication of how much internal work gets generated as a result of the type of requests the user is sending to Oracle.

user commits

This statistic is stored in V\$SYSSTAT. When a user commits a transaction, the redo generated that reflects the changes made to database blocks must be written to disk. Commits often represent the closest thing to a user transaction rate.

user rollbacks

This statistic is stored in V\$SYSSTAT. This statistic stores the number of times users manually issue the ROLLBACK statement or an error occurs during users' transactions.

write requests

This statistic is stored in V\$SYSSTAT. This statistic stores the number of time DBWR takes a batch of dirty buffers and writes them to disk.

C

Managing Partition Views

This section explains how to create and manage partition views, and includes the following topics:

- Partition View Guidelines
- Defining Partition Views
- Partition Views: Example

Partition View Guidelines

You may wish to create an index, reorganize the database, or perform other operations, only to discover that the resulting downtime may be too long for very large or mission-critical tables. One way to avoid significant downtime during operations is to create and use *partition views*.

You can use partition views by dividing very large tables into multiple, small pieces (partitions), which offer significant improvements in availability, administration and table scan performance. You create a partition view by dividing a large table into multiple physical tables using partitioning criteria. Then, for future queries, you can bring the table together as a whole. Also, you can use a key range to select from a partition view only the partitions that fall within that range.

Partition View Highlights

Partition views offer increased manageability and flexibility during queries. Individual partitions can be:

- added and dropped independently and efficiently
- reorganized, backed up and restored independently
- split, merged and loaded incrementally while maintaining local indexes. They can also be loaded in parallel.
- used in a SELECT clause (query or subquery)
- partitioned simultaneously by more than one column
- split, added or dropped online by replacing the view when ready to switch to a new configuration

Rules and Guidelines for Use

When creating and maintaining partition views, adhere to the following rules and guidelines:

- DDL commands must be issued separately for each underlying table.

For example, to add an index to a partition view, you must add indexes to all underlying tables. To analyze a partition view, you must analyze all underlying tables. However, you can submit operations on each partition in parallel.

- Administrative operations must be performed as operations on the underlying tables of the partition view, *not* on the partition view itself.

For example, a split operation consists of either one or two CREATE TABLE AS SELECT operations (one if the split is “in place”), followed by a redefining of the partition view’s view text.

- You can create referential integrity constraints on underlying tables, but for the constraints to be true for the partition view, both the primary and foreign keys must have the partition column as the first column.
- Similarly, you can have an unique index on underlying tables, but for uniqueness to be true for the partition view, the partition column must be the first column in the index. (It follows that you can have only one unique index.)
- Every partition has its own index, so any index lookup must be done in all indexes for partitions that are not skipped.
- A partition view cannot be the target of a DML statement.
- Partition views do not support concatenated partitioning keys.
- SQL*Loader does not support partition views.

Compared to non-partitioned tables, partition views should not add significant CPU overhead.

Partition Elimination

When a query contains a predicate that constrains the result set to a subset of a view's partitions, the optimizer chooses a plan that skips, or eliminates partitions that are not needed for the query. This *partition elimination* takes place at run time, when the execution plan references all partitions.

Note that the size of an execution plan is not reduced simply because partitions are skipped. In fact, the size of the execution plan is proportional to the number of partitions, and there is a practical upper limit on the number of partitions you can have: from a few dozen for tables with a large number of columns, to a few hundred for tables with a small number of columns. Also, even when partitions are skipped, there is a small amount of overhead (a fraction of a second) at run time per partition. Such overhead can be noticeable for a query that uses an index to retrieve a small number of records.

In the following example, a partitioned view on column C with the partitions P0, P1 and P2 has the following bounds:

```
P0.C between 0 and 9
P1.C between 10 and 19
P2.C between 20 and 29
```

Therefore the following query does not access the bulk of the blocks of P0 or P2:

```
SELECT * FROM partition_view WHERE C BETWEEN 12 and 15;
```


Defining Partition Views

Defining Partition Views Using Check Constraints

There are two ways to define partition views that will enable you to skip unneeded partitions:

- Defining Partition Views Using Check Constraints
- Defining Partition Views Using WHERE Clauses

Oracle Corporation recommends that you define partition views via the check constraint method, because:

- the check constraint predicates are not evaluated per row for queries
- the check constraint predicates guard against inserting rows into the wrong partition
- it is easier to query the data dictionary and find the partitioning criteria

The following example defines partition views for sales data over a calendar year:

```
ALTER TABLE Q1_SALES ADD CONSTRAINT C0 check (sale_date between
'Jan-01-1995' and 'Mar-31-1995');
ALTER TABLE Q2_SALES ADD CONSTRAINT C1 check (sale_date between
'Apr-01-1995' and 'Jun-30-1995');
ALTER TABLE Q3_SALES ADD CONSTRAINT C2 check (sale_date between
'Jul-01-1995' and 'Sep-30-1995');
ALTER TABLE Q4_SALES ADD CONSTRAINT C3 check (sale_date between
'Oct-01-1995' and 'Dec-31-1995');
CREATE VIEW sales AS
SELECT * FROM Q1_SALES UNION ALL
SELECT * FROM Q2_SALES UNION ALL
SELECT * FROM Q3_SALES UNION ALL
SELECT * FROM Q4_SALES;
```

Defining Partition Views Using WHERE Clauses

Alternatively, you can express the criteria in the WHERE clause of a view definition:

```
CREATE VIEW sales AS
SELECT * FROM Q1_SALES WHERE sale_date between
'Jan-01-1995' and 'Mar-31-1995' UNION ALL
SELECT * FROM Q2_SALES WHERE sale_date between
'Apr-01-1995' and 'Jun-30-1995' UNION ALL
SELECT * FROM Q3_SALES WHERE sale_date between
'Jul-01-1995' and 'Sep-30-1995' UNION ALL
SELECT * FROM Q4_SALES WHERE sale_date between
'Oct-01-1995' and 'Dec-31-1995';
```

Note: An advantage of using this method is that the partition view can be located at a remote database. However, this is not the recommended method for defining a partition view because the partitioning predicate is applied at runtime for all rows in all partitions that are not skipped. Also, if a user mistakenly inserts a row with `sale_date = 'Apr-04-1995'` in `Q1_SALES`, the row will “disappear” from the partition view. The partitioning criteria are also difficult to retrieve from the data dictionary because they are all embedded in one long view definition.

Partition Views: Example

This example shows how to:

- Create the Tables Underlying the Partition View
- Load Each Partition
- Enable Check Constraints
- Add Additional, Overlapping Partition Criteria
- Create Indexes for Each Partition
- Analyze the Partition View
- Create the View that Ties the Partitions Together

Create the Tables Underlying the Partition View

This example involves two tables, created with the following syntax:

```
create table line_item_1992 (
  constraint C_ship_date_1992
  check(ship_date between 'Jan-01-1992' and 'Dec-31-1992')
  disable,
  order_key          number ,
  part_key           number ,
  supp_key           number ,
  ship_date          date ,
  commit_date        date ,
  receipt_date       date );

create table line_item_1993 (
  constraint C_ship_date_1993
  check(ship_date between 'Jan-01-1993' and 'Dec-31-1993')
  disable,
  order_key          number ,
  part_key           number ,
  supp_key           number ,
  ship_date          date ,
  commit_date        date ,
  receipt_date       date );
```

Load Each Partition

You can load each partition using a SQL*Loader process. Each load process can reference the same loader control file (in this example it is "LI.ctl"), but should use a different data file. Also, the data files must match the partitioning criteria given in the ship_date check constraints.

```
sqlldr tpcd/tpcd direct=true control=LI.ctl data=LI1992.dat
sqlldr tpcd/tpcd direct=true control=LI.ctl data=LI1993.dat
```

Note: For improved performance, disable constraints that define the partitioning criteria until after the partitions are loaded.

Enable Check Constraints

After loading the partitions, you can enable the check constraints. Enabling the check constraints allows the optimizer to recognize and skip irrelevant partitions.

```
alter table line_item_1990 enable constraint C_ship_date_1992
alter table line_item_1991 enable constraint C_ship_date_1993
```

Add Additional Overlapping Partition Criteria

If you wish, you can have additional partitioning criteria or partitions that overlap. The table in this particular example is defined (ship_date + random(0..90)) so that the data generated does not contain any line item later than 90 days after it is shipped. You can utilize this information as follows:


```
alter table line_item_1992
  add constraint C2_1992 check ( receipt_date between
    'Jan-01-1992' and 'Jan-01-1993' + 90);

alter table line_item_1993
  add constraint C2_1993 check ( receipt_date between
    'Jan-01-1993' and 'Jan-01-1994' + 90);
```

Create Indexes for Each Partition

If you need an index on a partition view, you must create the index on each of the partitions. If you do not index each partition identically, the optimizer will be unable to recognize your UNION ALL view as a partition view.

```
create index part_key_supp_key_1992
  on line_item_1992 (part_key, supp_key)
create index part_key_supp_key_1993
  on line_item_1993 (part_key, supp_key)
```

Analyze the Partitions

Now analyze the partitions.

```
analyze table line_item_1992 compute statistics;
analyze table line_item_1993 compute statistics;
```

Note: You can submit an ANALYZE statement on each partition in parallel, using multiple logon sessions.

EXPLAIN PLAN Output

To confirm that the system recognizes your partition views, you must look at the following entries in the EXPLAIN PLAN output:

VIEW	This entry should include optimizer cost information.
UNION ALL	This entry should specify the option PARTITION for partition views.
FILTER	When an operation is a child of the UNION ALL operation, FILTER indicates that the underlying partition can be skipped.

These operations appear in the EXPLAIN PLAN output in the OPERATIONS column. The keyword PARTITION for UNION ALL operations appears in the OPTIONS column of the EXPLAIN PLAN. If PARTITION does not appear in this column, either of the following may have occurred:

- not all partitions have the same columns, in the same order, with the same data types as in the CREATE TABLE statements
- not all partitions have the same indexes, defined on the same columns, in the same order as in the CREATE INDEX statement

Example

The following sample EXPLAIN PLAN output indicates that the optimizer recognized that the query has a WHERE clause that limits the data returned from the partition view. Also, the FILTER indicates that the system recognized and used the check constraints to eliminate a partition. The keyword PARTITION in the UNION ALL line indicates that the system recognized that the underlying tables are the same shape and have the same indexes. Finally, the parallel information shows that the UNION ALL operation is executed in parallel.

```
explain plan for select * from line_item;
      where receipt_date = 'Feb-01-1992';

select substr (
  lpad (' ',2*(level-1))||decode(id,0,statement_id,operation)
||' '||options||' '||object_name, 1, 79) "step parallelinfo"
from plan_table
start with id = 0
connect by prior id = parent_id;
step parallelinfo

```

```
VIEW LINE_ITEM PARALLEL_TO_SERIAL
UNION_ALL PARTITION PARALLEL_COMBINED_WITH_PARENT
TABLE ACCESS FULL LINE_ITEM_1992 PARALLEL_COMBINED_WITH_PARENT
FILTER PARALLEL_COMBINED_WITH_PARENT
TABLE ACCESS FULL LINE_ITEM_1993 PARALLEL_COMBINED_WITH_PARENT
```

Create the View that Ties the Partitions Together

Once you identify or create the tables that you wish to use, you can create the view text that ties the partitions together.

```
create or replace view line_item as
  select * from line_item_1992 union all
  select * from line_item_1993;
```

Note: To keep users from accessing inconsistent or intermediate states of the partition view, this step has been deferred to a point after the data has been loaded, indexes have been built, and analyses are complete.

Partition Views and Parallelism

Partition views are scanned in parallel when all partitions are either skipped or accessed in parallel.

Partition constraints are for skipping only, not for allocating work to query server processes. The number of partitions is unrelated to the degree of parallelism. Full parallelism is used even if a single partition is not skipped.

D

Operating System–Specific Information

This manual occasionally refers to other Oracle manuals that contain detailed information for using Oracle on a specific operating system. These Oracle manuals are generically referred to as operating system–specific Oracle documentation, as the exact name varies on different operating systems.

This appendix lists all the references in this addendum to operating system–specific Oracle manuals. If you are using Oracle on multiple operating systems, this appendix can help you ensure that your applications are portable across these operating systems.

Operating system–specific topics are listed alphabetically with page numbers of sections that discuss these topics.

Topics

File Striping, 6 – 14

I/O Monitoring and Tuning, 9 – 2

Memory Tuning, 8 – 3

Monitoring CPU, 6 – 15

Tuning Your Operating System, 1 – 6

Parallel Query Tuning, 6 – 13

Index

A

- access paths, 5 – 4
 - bounded range on indexed columns, 5 – 37
 - cluster join, 5 – 34
 - composite index, 5 – 36
 - full table scan, 5 – 41
 - hash cluster key, 5 – 35
 - indexed cluster key, 5 – 35
 - listed, 5 – 31
 - MAX or MIN of indexed column, 5 – 40
 - optimization, 5 – 29
 - ORDER BY on indexed column, 5 – 40
 - single row by cluster join, 5 – 32
 - single row by hash cluster key
 - (with unique key), 5 – 33
 - single row by ROWID, 5 – 32
 - single row by unique or primary key, 5 – 34
 - single-column index, 5 – 36
 - sort-merge join, 5 – 39
 - unbounded range on indexed columns, 5 – 38
- adding, dispatcher processes, 10 – 6
- ALL, 5 – 12
- ALL_INDEXES view, 7 – 40
- ALL_ROWS hint, 5 – 29, 7 – 16
- allocating, extents, 9 – 11
- allocation, of memory. *See* memory allocation
- ALTER SESSION command
 - examples, A – 8
 - SET SESSION_CACHED_CURSORS, 8 – 14
- ANALYZE command, 9 – 8
 - examples, 7 – 12

- AND_EQUAL hint, 7 – 6, 7 – 25
- anonymous PL/SQL blocks
 - application uses, 2 – 11 to 2 – 12
 - performance benefits, 2 – 10
- ANY, 5 – 12
- applications
 - client/server, 2 – 10 to 2 – 13
 - decision support, 2 – 3 to 2 – 4, 6 – 4, 6 – 13
 - distributed databases, 2 – 5 to 2 – 6
 - multi-purpose, 2 – 8 to 2 – 10
 - OLTP, 2 – 2 to 2 – 3
 - parallel query option, 2 – 7 to 2 – 8
 - Parallel Server, 2 – 7
 - registering with the database, 3 – 2 to 3 – 5
 - scientific or statistical, 2 – 4
 - using PL/SQL, 2 – 12
- array processing, 4 – 6
- assigning, rollback segments to transactions, 9 – 12

B

- BEGIN_DISCRETE_TRANSACTION
 - procedure, 4 – 8
- BETWEEN, 5 – 13
- bind variables, 4 – 6
 - optimization, 5 – 45 to 5 – 46
 - shared SQL areas, 8 – 12
- bitmap index, 7 – 34 to 7 – 37, 7 – 40, 7 – 41
- bottlenecks
 - disk I/O, 9 – 2

- identifying, 1 – 7, A – 2 to A – 4
- memory, 8 – 2
- buffer cache
 - memory allocation, 8 – 19 to 8 – 24
 - reducing cache misses, 8 – 19
 - tuning, 8 – 18 to 8 – 24
- buffers
 - determining number to add, 8 – 21
 - when to reduce number of, 8 – 21
- BUSY column, V\$DISPATCHER table, 10 – 4

C

- CACHE hint, 7 – 29
- cartesian products, 5 – 10
- chained rows, 9 – 8
- Checkpoint process (CKPT)
 - behavior on checkpoints, 11 – 9
 - enabling, 11 – 9
- CHECKPOINT_PROCESS parameter, 11 – 9
- checkpoints
 - choosing checkpoint frequency, 11 – 9
 - effect on recovery time performance, 11 – 8
 - effect on runtime performance, 11 – 8
 - redo log maintenance, 11 – 9
 - signalling DBWR to write, 11 – 8
 - tuning, 11 – 8
- CHOOSE, 5 – 29
- CHOOSE hint, 7 – 18
- client/server applications, 2 – 10
- CLUSTER hint, 7 – 19
- cluster joins, 5 – 49 to 5 – 51
- clusters
 - hash
 - scans, 5 – 30
 - searches, 5 – 33, 5 – 35
 - how to use, 7 – 7
 - index, searches on, 5 – 35
 - scans, 5 – 29
 - tradeoffs, 7 – 7
- columns, choosing for indexes, 7 – 3
- COMPATIBLE parameter, 7 – 40
- composite indexes, 7 – 4

- consistent gets statistic, 8 – 18, 10 – 3, 11 – 8
- calculating hit ratio, 8 – 20, 8 – 23
- consistent mode, A – 17
- contention
 - disk access, 6 – 14, 9 – 2
 - free lists, 11 – 7
 - memory, 8 – 2
 - memory access, 10 – 1
 - parallel query option, 6 – 14
 - redo allocation latch, 10 – 13
 - redo copy latches, 10 – 13
 - rollback segments, 10 – 2 to 10 – 3
 - tuning, 10 – 1
 - V\$SESSION_WAIT, 1 – 7, A – 2
- controlling transactions, 4 – 7
- cost-based optimization, 5 – 6, 7 – 11
- COUNT column
 - X\$KCBCBH table, 8 – 22
 - X\$KCBRBH table, 8 – 19
- count column, SQL trace facility output, A – 16
- cpu column, SQL trace facility output, A – 16
- CREATE CLUSTER command, 7 – 9
- CREATE INDEX command
 - examples, 11 – 4
 - NOSORT option, 11 – 4
- CREATE TABLE command
 - examples, 9 – 5 to 9 – 6
 - parallelism, 6 – 4
 - STORAGE clause, 9 – 5
 - syntax, 6 – 10
 - TABLESPACE clause, 9 – 5 to 9 – 7
- CREATE TABLESPACE command
 - DATAFILE clause, 9 – 5 to 9 – 6
- cross joins, 5 – 10
- current column, SQL trace facility output,
 - A – 17
- current mode, number of buffers retrieved,
 - A – 17
- CURSOR_NUM column, TKPROF_TABLE,
 - A – 20
- CURSOR_SPACE_FOR_TIME parameter,
 - setting, 8 – 13
- cursors, creating, 4 – 4

D

- data definition statements (DDL), processing, 4 – 7
- data dictionary, views used in optimization, 5 – 7
- data dictionary cache
 - reducing cache misses, 8 – 16
 - tuning, 8 – 14
- data manipulation statements (DML), processing, 4 – 4
- data warehousing, star queries, 5 – 55
- Database Writer process (DBWR), behavior on checkpoints, 11 – 8
- databases, distributed, statement optimization, 5 – 61
- DATAFILE clause
 - CREATE TABLESPACE command, 9 – 5 to 9 – 6
 - examples, 9 – 5 to 9 – 6
- datafiles
 - parallel query option and, 6 – 14
 - placement on disk, 9 – 4
- DATE_OF_INSERT column, TKPROF_TABLE, A – 20
- db block gets statistic, 8 – 18, 10 – 3, 11 – 8
 - calculating hit ratio, 8 – 20, 8 – 23
- DB_BLOCK_BUFFERS parameter
 - reducing buffer cache misses, 8 – 19
 - removing unneeded buffers, 8 – 21
- DB_BLOCK_LRU_EXTENDED_STATISTICS parameter, setting, 8 – 19
- DB_BLOCK_LRU_STATISTICS parameter, setting, 8 – 22
- DB_FILE_MULTIBLOCK_READ_COUNT parameter
 - cost-based optimization, 5 – 54
 - optimization and, 5 – 44
- DBA_INDEXES view, 7 – 40
- DBMS_APPLICATION_INFO package, 3 – 2, 3 – 4
- DBMS_SHARED_POOL package, 4 – 14 to 4 – 17
- DBMS_SYSTEM package, A – 9
- DBMSPOOL.SQL script, 4 – 14
- DBMSUTL.SQL, 3 – 3
- decision support, 2 – 3 to 2 – 4, 6 – 4
 - star queries, 5 – 55
 - tuning, 6 – 13
 - with OLTP, 2 – 8
- define phase of query processing, 4 – 6
- degree of parallelism, 6 – 3, 6 – 9
 - between operations, 6 – 7
 - combining operations, 6 – 19
 - EXPLAIN PLAN command, 6 – 18
 - hints, 6 – 10
 - setting, 6 – 9, 6 – 15
- DEPTH column, TKPROF_TABLE, A – 20
- describe phase of query processing, 4 – 6
- designing and tuning, 1 – 5 to 1 – 6
- discrete transactions, 4 – 8 to 4 – 11
 - errors, 4 – 9
 - example, 4 – 10
 - processing, 4 – 9
 - usage notes, 4 – 9
 - when to use, 4 – 8
- disk column, SQL trace facility output, A – 17
- disks
 - avoiding contention, 9 – 4
 - contention, 9 – 2
 - parallel query option, 6 – 14
 - distributing I/O, 9 – 4
 - monitoring I/O, 9 – 2 to 9 – 3
 - monitoring OS file activity, 9 – 3
 - placement of datafiles, 9 – 4
 - placement of redo log files, 9 – 4
 - striping
 - automatically, 6 – 14
 - manually, 9 – 5
- dispatcher processes (Dnnn), adding, 10 – 6
- distributed databases, 2 – 5 to 2 – 6
 - statement optimization, 5 – 61 to 5 – 64
- distributed processing environment, data manipulation statements, 4 – 4
- distributing I/O, 9 – 4 to 9 – 7
 - parallel query option, 6 – 14
- DIUTIL package, 4 – 14

dynamic extension, 9 – 10
 avoiding, 9 – 11 to 9 – 13
dynamic performance tables, enabling
 statistics, A – 8

E

elapsed column, SQL trace facility output,
 A – 16
enabling
 Checkpoint process (CKPT), 11 – 9
 SQL trace facility, A – 8 to A – 10
equijoins, defined, 5 – 10
errors, during discrete transactions, 4 – 9
examples
 ALTER SESSION command, A – 8
 ANALYZE command, 7 – 12
 CREATE TABLESPACE command, 9 – 5 to
 9 – 6
 CREATE INDEX command, 11 – 4
 CREATE TABLE command, 9 – 5 to 9 – 6
 DATAFILE clause, 9 – 5 to 9 – 6
 discrete transactions, 4 – 10
 execution plan, 7 – 30
 EXPLAIN PLAN output, 6 – 18, 7 – 30,
 A – 15, A – 27
 full table scan, 7 – 30
 indexed query, 7 – 30
 NOSORT option, 11 – 4
 SET TRANSACTION command, 9 – 12
 SQL trace facility output, A – 15
 STORAGE clause, 9 – 5
 table striping, 9 – 5
 TABLESPACE clause, 9 – 5 to 9 – 7
execution plans, A – 21
 examples, 5 – 18 to 5 – 26, 7 – 30, A – 16
 execution sequence of, 5 – 4
 overview of, 5 – 2 to 5 – 5
 TKPROF, A – 11, A – 16
 viewing, 5 – 5
EXPLAIN PLAN command, A – 21
 about, 6 – 18
 examples of output, 6 – 18, 7 – 30, A – 15
 exapmles of output, A – 27
 invoking with the TKPROF program, A – 11
PLAN_TABLE, A – 22

extension, generates recursive calls, 9 – 10
extents, allocating to avoid extension, 9 – 11

F

FAST FULL SCAN, 5 – 30
fetching rows in a query, 4 – 7
FIRST_ROWS hint, 5 – 29, 7 – 16
free lists
 adding, 11 – 8
 contention, 11 – 7
 reducing contention, 11 – 8
FREELIST GROUPS parameter, STORAGE
 clause, 9 – 7
FREELISTS parameter, STORAGE clause, 9 – 7
FULL hint, 7 – 6, 7 – 19
full table scans, 5 – 29, 5 – 41, 6 – 3
 example, 7 – 30
 parallel query option, 6 – 2 to 6 – 4

G

GETMISSES column, V\$ROWCACHE table,
 8 – 15, 8 – 16
GETS column
 V\$LATCH table, 10 – 11
 V\$ROWCACHE table, 8 – 15 to 8 – 16
goals for tuning, 1 – 4 to 1 – 5
GROUP BY NOSORT, 11 – 4

H

HASH hint, 7 – 19
hash join, 5 – 50
HASH parameter, CREATE CLUSTER
 command, 7 – 9
hashing, how to use, 7 – 8
HASHKEYS parameter, CREATE CLUSTER
 command, 7 – 9
HIGH_VALUE column,
 of USER_TAB_COLUMNS, 5 – 45

- hints, 7 – 14
 - access methods, 7 – 18
 - ALL_ROWS, 7 – 16
 - AND_EQUAL, 7 – 6, 7 – 25
 - CACHE, 7 – 29
 - CHOOSE, 7 – 18
 - CLUSTER, 7 – 19
 - degree of parallelism, 6 – 10, 7 – 28 to 7 – 30
 - FIRST_ROWS, 7 – 16
 - FULL, 7 – 6, 7 – 19
 - HASH, 7 – 19
 - how to use, 7 – 14
 - INDEX, 5 – 57, 7 – 6, 7 – 20
 - INDEX_ASC, 7 – 21
 - INDEX_DESC, 7 – 22
 - INDEX_FFS, 5 – 31
 - join operations, 7 – 26
 - join orders, 7 – 25
 - NOCACHE, 7 – 29
 - NOPARALLEL hint, 7 – 28
 - optimization approach and goal, 7 – 16
 - ORDERED, 5 – 54, 5 – 57, 7 – 25
 - PARALLEL hint, 7 – 28
 - parallel query option, 7 – 28 to 7 – 30
 - PUSH_SUBQ, 7 – 29
 - ROWID, 7 – 19
 - RULE, 7 – 18
 - STAR, 5 – 57
 - USE_CONCAT, 7 – 25
 - USE_MERGE, 7 – 27
 - USE_NL, 7 – 26
- HOLD_CURSOR, 8 – 7

I

- I/O
 - distributing, 9 – 4 to 9 – 7
 - parallel query option, 6 – 14
 - tuning, 9 – 2
- ID column, PLAN_TABLE table, A – 24
- IDLE column, V\$DISPATCHER table, 10 – 4
- IN, 5 – 12
- INDEX hint, 5 – 31, 7 – 6, 7 – 20, 7 – 40
- INDEX_ASC hint, 7 – 21
- INDEX_COMBINE hint, 7 – 40
- INDEX_DESC hint, 7 – 22

- INDEX_FFS hint, 5 – 31
- indexes
 - avoiding the use of, 7 – 6
 - bitmap, 7 – 34 to 7 – 37, 7 – 40, 7 – 41
 - choosing columns for, 7 – 3
 - composite, 5 – 36, 7 – 4
 - creating in parallel, 6 – 20
 - ensuring the use of, 7 – 5
 - example, 7 – 30
 - FAST FULL SCAN, 5 – 30
 - how to use, 7 – 2
 - modifying values of, 7 – 3
 - optimization and, 5 – 15 to 5 – 20
 - parallel creation, 6 – 20
 - placement on disk, 9 – 6
 - range scans, 5 – 30
 - scans, 5 – 30
 - searches on, 5 – 36 to 5 – 39
 - using MAX or MIN, 5 – 40
 - using ORDER BY, 5 – 40
 - selectivity of, 7 – 3
 - statement conversion and, 5 – 15 to 5 – 18
 - STORAGE clause with parallel query
 - option, 6 – 20
 - unique scans, 5 – 30
- INDX column
 - X\$KCBCBH table, 8 – 22
 - X\$KCBRBH table, 8 – 19
- INITIAL parameter, STORAGE clause, 9 – 7
- initialization parameters
 - DISCRETE_TRANSACTIONS_ENABLED, 4 – 9
 - MAX_DUMP_FILE_SIZE, A – 8
 - OPTIMIZER_MODE, 5 – 27, 7 – 14, 7 – 16
 - PRE_PAGE_SGA, 8 – 4
 - SESSION_CACHED_CURSORS, 8 – 14
 - SORT_DIRECT_WRITES, 6 – 16, 11 – 5
 - SORT_WRITE_BUFFER_SIZE, 11 – 5
 - SORT_WRITE_BUFFERS, 11 – 5
 - SQL_TRACE, A – 9
 - TIMED_STATISTICS, A – 8
 - USER_DUMP_DEST, A – 8
- instances, limiting for parallel queries, 6 – 11
- inter-operator parallelism, 6 – 7
- intra-operator parallelism, 6 – 7
- isolation level, 4 – 12

J

- joins, 5 – 9
 - cartesian products, 5 – 10
 - cluster, 5 – 32, 5 – 49 to 5 – 51
 - searches on, 5 – 34
 - convert to subqueries, 5 – 17 to 5 – 20
 - cross, 5 – 10
 - equijoins, 5 – 10
 - execution plans and, 5 – 47
 - hash, 5 – 50
 - nested loops, 5 – 47 to 5 – 49
 - cost-based optimization, 5 – 54
 - nonequijoins, 5 – 10
 - optimization of, 5 – 52
 - outer, 5 – 10
 - parallel query option, 6 – 6
 - sort-merge, 5 – 48 to 5 – 50
 - cost-based optimization, 5 – 54
 - sort-merge searches, 5 – 39

K

- KEEP procedure, 4 – 15
- keys, searches, 5 – 33

L

- latches
 - redo allocation latch, 10 – 9
 - redo copy latches, 10 – 9
- library cache
 - memory allocation, 8 – 11
 - tuning, 8 – 9
- LIKE, 5 – 11
- log. *See* redo log
- log switches, tuning checkpoints, 11 – 9
- Log Writer process (LGWR), tuning, 9 – 4
- LOG_BUFFER parameter, setting, 10 – 9
- LOG_CHECKPOINT_INTERVAL parameter, guidelines, 11 – 9
- LOG_CHECKPOINT_TIMEOUT parameter, guidelines, 11 – 9

- LOG_SIMULTANEOUS_COPIES parameter, 10 – 13
 - setting, 10 – 10
- LOG_SMALL_ENTRY_MAX_SIZE parameter, setting, 10 – 10, 10 – 13
- LOW_VALUE column, of USER_TAB_COLUMNS, 5 – 45

M

- Managment Information Base (MIB).
See Simple Network Management Protocol (SNMP)
- MAX operator, 5 – 40
- max session memory statistic, 8 – 16
- MAX_DUMP_FILE_SIZE, A – 8
- MAXEXTENTS parameter, STORAGE clause, 9 – 7
- MAXOPENCURSORS, 8 – 7
- memory allocation
 - buffer cache, 8 – 19 to 8 – 24
 - importance, 8 – 2
 - library cache, 8 – 11
 - shared SQL areas, 8 – 11
 - sort areas, 11 – 2
 - tuning, 8 – 2, 8 – 24
 - users, 8 – 5
- migrated rows, 9 – 8
- MIN operator, 5 – 40
- MINEXTENTS parameter, STORAGE clause, 9 – 7
- mirroring, redo log files, 9 – 4
- MISSES column, V\$LATCH table, 10 – 11
- monitoring the system, 1 – 8
- MTS_MAX_DISPATCHERS parameter, tuning dispatchers, 10 – 6
- MTS_MAX_SERVERS parameter, tuning servers, 10 – 7
- multi-block reads, 9 – 11
- multi-purpose applications, 2 – 8

- multi-threaded server
 - reducing contention, 10 – 4
 - shared pool and, 8 – 16
 - tuning, 10 – 4 to 10 – 7

N

- NAMESPACE column, V\$LIBRARYCACHE table, 8 – 9
- nested loops joins, 5 – 47 to 5 – 49
 - cost-based optimization, 5 – 54
- networks, minimizing traffic, 2 – 10
- NEXT parameter, STORAGE clause, 9 – 7
- NLS_SORT parameter, ORDER BY access path, 5 – 41
- NOCACHE hint, 7 – 29
- nonequijoins, 5 – 10
- NONUNIQUE index, 7 – 40
- NOPARALLEL hint, 7 – 28
- NOSORT option, 11 – 4
 - choosing when to use, 11 – 4
 - CREATE INDEX command, 11 – 4
 - examples, 11 – 4
 - GROUP BY, 11 – 4
 - performance benefits, 11 – 3 to 11 – 4
- NUM_DISTINCT column, of USER_TAB_COLUMNS, 5 – 45
- NUM_ROWS column, of USER_TABLES view, 5 – 45

O

- OBJECT_INSTANCE column, PLAN_TABLE table, A – 23
- OBJECT_NAME column, PLAN_TABLE table, A – 23
- OBJECT_NODE column, PLAN_TABLE table, A – 23
- OBJECT_OWNER column, PLAN_TABLE table, A – 23
- OBJECT_TYPE column, PLAN_TABLE table, A – 24
- online redo log, increasing size, 11 – 9

- online transaction processing (OLTP), 2 – 2 to 2 – 3
 - rollback segment extension, 9 – 12
 - with decision support, 2 – 8
- OPEN_CURSORS parameter
 - allocating more private SQL areas, 8 – 7
 - increasing cursors per session, 8 – 11
- operating system
 - monitoring disk I/O, 9 – 2, 9 – 3
 - tuning, 1 – 6, 8 – 3
- OPERATION column
 - PLAN_TABLE table, A – 23
 - values, A – 25
- operators
 - MAX, 5 – 40
 - MIN, 5 – 40
- OPTIMAL parameter, STORAGE clause, 9 – 7
- OPTIMAL storage parameter, 9 – 12
- optimization, 5 – 2
 - choosing an approach and goal for, 7 – 11
 - choosing the approach, 5 – 27
 - conversion of expressions and predicates, 5 – 11
 - cost-based, 5 – 6, 5 – 54 to 5 – 56
 - choosing an access path, 5 – 43 to 5 – 45
 - examples of, 5 – 44 to 5 – 47
 - remote databases and, 5 – 61
 - when to use, 7 – 11
 - hints, 5 – 29
 - manual, 5 – 29
 - rule-based, 5 – 6, 5 – 52
 - choosing an access path, 5 – 42 to 5 – 44
 - examples of, 5 – 42 to 5 – 44
 - when to use, 7 – 14
 - selectivity of queries and, 5 – 44
 - transitivity and, 5 – 13
- optimizer, 5 – 2
 - parallel queries, 6 – 5
 - when used, 4 – 16
- OPTIMIZER column, PLAN_TABLE, A – 24
- OPTIMIZER_GOAL option, 5 – 28
 - ALTER SESSION command, 7 – 13
 - of ALTER SESSION command, 7 – 11

- OPTIMIZER_MODE, 5 – 27, 7 – 11, 7 – 14, 7 – 16
 - hints affecting, 5 – 29
- OPTIONS column, PLAN_TABLE table, A – 23
- Oracle Call Interface (OCI)
 - bind variables, 4 – 6
 - control of parsing and private SQL areas, 8 – 7
- Oracle Forms, A – 9
 - control of parsing and private SQL areas, 8 – 8
 - PL/SQL, 2 – 12
 - PL/SQL engine, 2 – 11
- Oracle Parallel Server. *See* Parallel Server
- Oracle Precompilers
 - bind variables, 4 – 6
 - control of parsing and private SQL areas, 8 – 7
 - PL/SQL, 2 – 12
- Oracle Server
 - client/server configuration, 2 – 10
 - configurations, 2 – 5 to 2 – 9
 - PL/SQL engine, 2 – 11
 - SQL processing, 4 – 4
 - transactions, 4 – 7
- Oracle Server Manager. *See* Server Manager
- Oracle Trace, A – 30
- ORDERED hint, 5 – 54, 7 – 25
- OTHER column, PLAN_TABLE table, A – 24
- outer joins, 5 – 10
- overloaded disks, 9 – 4

P

- packages
 - DBMS_APPLICATION_INFO, 3 – 2, 3 – 4
 - DBMS_SHARED_POOL, 4 – 14
 - DBMS_TRANSACTION, 4 – 10 to 4 – 11
 - DIUTIL, 4 – 14
 - performance benefits, 2 – 11
 - registering with the database, 3 – 2
 - STANDARD, 4 – 14
- paging
 - library cache, 8 – 11
 - reducing, 8 – 4

- SGA, 8 – 24
- PARALLEL hint, 7 – 28
- parallel query option, 2 – 7 to 2 – 8, 6 – 2
 - combining operations, 6 – 19
 - degree of parallelism, 6 – 9
 - setting, 6 – 15
 - EXPLAIN PLAN command, 6 – 18
 - full table scans, 6 – 2
 - hints, 7 – 28
 - index creation, 6 – 20
 - inter-operator parallelism, 6 – 7
 - intra-operator parallelism, 6 – 7
 - multi-threaded server, 6 – 4
 - number of server processes, 6 – 12
 - operations in execution plan, 6 – 6
 - optimizer, 6 – 5
 - Parallel Server and, 6 – 1, 6 – 11
 - partitioning rows, 6 – 6
 - query coordinator, 6 – 3
 - query servers, 6 – 12, 10 – 8
 - SORT_DIRECT_WRITES parameter, 6 – 16
 - summary or rollup tables, 6 – 4
 - tuning, 6 – 13 to 6 – 20
 - tuning query servers, 6 – 17, 10 – 8
- Parallel Server, 2 – 7
 - parallel query option, 6 – 1, 6 – 11
- PARALLEL_MAX_SERVERS parameter, 6 – 12
- PARALLEL_MIN_SERVERS parameter, 6 – 12
- PARALLEL_SERVER_IDLE_TIME parameter, 6 – 12
- PARAMETER column,
 - V\$ROWCACHE table, 8 – 15
- PARENT_ID column,
 - PLAN_TABLE table, A – 24
- parsing, 4 – 4
 - Oracle Call Interface (OCI), 8 – 7
 - Oracle Forms, 8 – 8
 - Oracle Precompilers, 8 – 7
 - reducing unnecessary calls, 8 – 6 to 8 – 8
 - SQL statements, 4 – 4
- PCTINCREASE parameter,
 - STORAGE clause, 9 – 7
- performance
 - bottlenecks, 1 – 7, A – 2
 - client/server applications, 2 – 10
 - decision support applications, 2 – 3

- different types of applications, 2 – 2 to 2 – 5
 - distributed databases, 2 – 5
 - monitoring registered applications, 3 – 2
 - multi-purpose applications, 2 – 8
 - OLTP applications, 2 – 2
 - packages benefits, 2 – 11
 - Parallel Server, 2 – 7
 - PL/SQL benefits, 2 – 10
 - scientific applications, 2 – 4
 - stored procedures benefits, 2 – 10
 - viewing execution plans, 5 – 5
 - PHYRDS column, V\$FILESTAT table, 9 – 3
 - physical reads statistic, 8 – 18
 - calculating hit ratio, 8 – 20, 8 – 23
 - PHYWRFS column, V\$FILESTAT table, 9 – 3
 - PINS column,
 - V\$LIBRARYCACHE table, 8 – 10
 - PL/SQL, 2 – 10
 - anonymous blocks, 2 – 10
 - application uses, 2 – 12
 - location of PL/SQL engine, 2 – 11
 - Oracle Forms, 2 – 12
 - Oracle Precompilers, 2 – 12
 - performance benefits, 2 – 10
 - SQL*Plus, 2 – 12
 - stored procedures, 2 – 10
 - tuning PL/SQL areas, 8 – 6
 - PLAN_TABLE table
 - ID column, A – 24
 - OBJECT_INSTANCE column, A – 23
 - OBJECT_NAME column, A – 23
 - OBJECT_NODE column, A – 23
 - OBJECT_OWNER column, A – 23
 - OBJECT_TYPE column, A – 24
 - OPERATION column, A – 23
 - OPTIMIZER column, A – 24
 - OPTIONS column, A – 23
 - OTHER column, A – 24
 - PARENT_ID column, A – 24
 - POSITION column, A – 24
 - REMARKS column, A – 23
 - SEARCH_COLUMNS column, A – 24
 - STATEMENT_ID column, A – 23
 - structure, A – 22
 - TIMESTAMP column, A – 23
 - POSITION column,
 - PLAN_TABLE table, A – 24
 - PRE_PAGE_SGA parameter, 8 – 4
 - primary keys
 - optimization, 5 – 18
 - searches, 5 – 34
 - private SQL areas, reuse by multiple SQL
 - statements, 8 – 6
 - procedures. *See* stored procedures
 - process priority, 10 – 14
 - PROCESSES parameter,
 - increasing for CKPT, 11 – 9
 - processing
 - distributed processing environment, 2 – 10, 4 – 5
 - queries, 4 – 5
 - production systems, tuning, 1 – 6 to 1 – 7
 - PUSH_SUBQ hint, 7 – 29
- ## Q
- queries, 5 – 9
 - ad hoc, 6 – 4
 - avoiding the use of indexes, 7 – 6
 - compound, 5 – 10
 - optimization of, 5 – 58 to 5 – 62
 - ORs converted to, 5 – 15
 - define phase, 4 – 6
 - describe phase, 4 – 6
 - ensuring the use of indexes, 7 – 5
 - fetching rows, 4 – 5
 - parallel processing, 6 – 2
 - processing, 4 – 5
 - selectivity, 5 – 44
 - star queries, 5 – 55
 - query column, SQL trace facility output, A – 17
 - query coordinator, 6 – 3
 - query plan. *See* execution plan
 - query server process, 6 – 3
 - about, 6 – 3, 6 – 12
 - tuning, 6 – 17, 10 – 8

R

- recovery from instance failure, effect of
 - checkpoints, 11 – 8
- recursive calls, 9 – 10
 - detected by the SQL trace facility, A – 17
 - dynamic extension, 9 – 10
 - statistic, 9 – 10
- recursive SQL, shared SQL, 4 – 13
- redo allocation latch, 10 – 10
 - contention, 10 – 13
- redo copy latches, 10 – 10
 - choosing how many, 10 – 10, 10 – 13
 - contention, 10 – 13
 - creating more, 10 – 13
- redo log files
 - mirroring, 9 – 4
 - placement on disk, 9 – 4
 - tuning checkpoints, 11 – 9
- redo log space requests statistic, 10 – 9
- reducing
 - buffer cache misses, 8 – 19
 - contention
 - dispatchers, 10 – 4
 - OS processes, 10 – 14
 - query servers, 10 – 8
 - redo log buffer latches, 10 – 9 to 10 – 13
 - shared servers, 10 – 6
 - data dictionary cache misses, 8 – 16
 - disk contention, 9 – 2
 - library cache misses, 8 – 10
 - network traffic, 2 – 10
 - number of database buffers, 8 – 21
 - paging and swapping, 8 – 4
 - query execution time
 - optimizer, 5 – 2
 - parallel query option, 6 – 2
 - rollback segment contention, 10 – 3
 - unnecessary parse calls, 8 – 6
- registering applications with database, 3 – 2 to 3 – 5
- RELEASE_CURSOR, 8 – 7
- RELOADS column,
 - V\$LIBRARYCACHE table, 8 – 10
- REMARKS column,
 - PLAN_TABLE table, A – 23

- response time, 5 – 7
 - cost-based approach, 5 – 27 to 5 – 29
 - optimizing, 7 – 12 to 7 – 13, 7 – 16
- rollback segments
 - assigning to transactions, 9 – 12
 - choosing how many, 10 – 3
 - contention, 10 – 2 to 10 – 3
 - creating, 10 – 3
 - detecting dynamic extension, 9 – 10
 - dynamic extension, 9 – 12
 - OLTP applications, 9 – 12
- row sources, 5 – 3 to 5 – 5
- ROWID hint, 7 – 19
- ROWIDs, table access by, 5 – 29
- rows
 - fetches, 4 – 5
 - row sources, 5 – 3 to 5 – 5
 - ROWIDs used to locate, 5 – 29, 5 – 32
- rows column, SQL trace facility output, A – 17
- RULE hint, 7 – 18
 - OPTIMIZER_MODE and, 5 – 29
- rule-based optimization, 5 – 6, 7 – 14

S

- scans, 5 – 29
 - bounded range on indexed columns, 5 – 37
 - cluster, 5 – 29, 5 – 34
 - FAST FULL SCAN, 5 – 30
 - full table, 5 – 29, 5 – 41
 - parallel query option, 6 – 2 to 6 – 4
 - hash, 5 – 30, 5 – 35
 - index, 5 – 30, 5 – 36, 5 – 40
 - range, 5 – 30
 - unbounded range on indexed columns,
 - 5 – 38
 - unique, 5 – 30
- scientific applications, 2 – 4 to 2 – 5
- SEARCH_COLUMN column,
 - PLAN_TABLE table, A – 24
- segments, 9 – 10
- selectivity
 - indexes, 7 – 3
 - queries, 5 – 44

- SERIALIZABLE, 4 – 12
- serializable transactions, 4 – 12
- Server Manager, 1 – 8
 - SHOW SGA command, 8 – 4
- session memory statistic, 8 – 16
- SESSION_CACHED_CURSORS parameter, 8 – 14
- SET TRANSACTION command
 - assigning transactions to rollback segments, 9 – 12
 - examples, 9 – 12
- shared pool
 - keeping objects pinned in, 4 – 14
 - tuning, 8 – 8 to 8 – 17
- shared SQL areas
 - finding large areas, 4 – 15
 - identical SQL statements, 4 – 13
 - keeping in the shared pool, 4 – 14 to 4 – 17
 - memory allocation, 8 – 11
 - statements considered, 4 – 13
- SHARED_POOL_SIZE parameter
 - allocating library cache, 8 – 11
 - reducing dictionary cache misses, 8 – 16
 - tuning the shared pool, 8 – 16
- SHOW SGA command, 8 – 4
- Simple Network Management Protocol (SNMP), 1 – 8
 - Management Information Base (MIB), 1 – 8
- SIZES procedure, 4 – 15
- SLEEPS column, V\$LATCH table, 10 – 11
- SNMP. *See* Simple Network Management Protocol (SNMP)
- SOME, 5 – 12
- sort areas
 - memory allocation, 11 – 2
 - memory required, 6 – 16
- sort-merge joins, 5 – 48 to 5 – 50
 - cost-based optimization, 5 – 54
- SORT_AREA_RETAINED_SIZE parameter, tuning sorts, 11 – 3
- SORT_AREA_SIZE parameter, 6 – 16, 7 – 39
 - cost-based optimization and, 5 – 54
 - tuning sorts, 11 – 2
- SORT_DIRECT_WRITES parameter, 11 – 5
 - parallel query option, 6 – 16
- SORT_WRITE_BUFFERS, 11 – 5
- sorts
 - avoiding on index creation, 11 – 3
 - parallel query option, 6 – 6
 - tuning, 11 – 2
- sorts (disk) statistic, 11 – 2
- sorts (memory) statistic, 11 – 2
- SQL areas, tuning, 8 – 6
- SQL statements, 4 – 2
 - array processing, 4 – 6
 - avoiding the use of indexes, 7 – 6
 - binding variables, 4 – 6
 - complex, 5 – 10
 - converting, examples of, 5 – 14 to 5 – 18
 - creating cursors, 4 – 4
 - distributed
 - defined, 5 – 11
 - optimization, 5 – 61 to 5 – 64
 - ensuring the use of indexes, 7 – 5
 - execution, 4 – 2, 4 – 6
 - execution plans, 5 – 2
 - modifying indexed data, 7 – 3
 - optimization of, 5 – 9 to 5 – 11
 - optimization of complex, 5 – 17 to 5 – 19
 - parallel query option, 6 – 2
 - parallelizing, 6 – 5
 - parsing, 4 – 4
 - recursive, OPTIMIZER_GOAL does not affect, 5 – 28
 - simple, 5 – 9
 - transactions, 4 – 7
 - tuning, 7 – 1
- SQL trace facility, A – 7
 - See also* TKPROF program
 - enabling, A – 8 to A – 10
 - example of output, A – 15
 - output, A – 16 to A – 18
 - parse calls, 8 – 6
 - steps to follow, A – 7
 - trace files, A – 8, A – 10
- SQL*Plus, PL/SQL, 2 – 12
- SQL_STATEMENT column, TKPROF_TABLE, A – 20
- SQL_TRACE parameter, A – 9
- STANDARD package, 4 – 14
- STAR hint, 5 – 57

- star query
 - environment, 5 – 55
 - extended star schemas, 5 – 56
 - tuning, 5 – 55, 5 – 56
- STATEMENT_ID column,
 - PLAN_TABLE table, A – 23
- statistical systems, 2 – 4 to 2 – 5
- statistics
 - consistent gets, 8 – 18, 10 – 3, 11 – 8
 - db block gets, 8 – 18, 10 – 3
 - dispatcher processes, 10 – 4
 - enabling collection, 8 – 19
 - max session memory, 8 – 16
 - optimizer use of, 5 – 6 to 5 – 8
 - physical reads, 8 – 18
 - query servers, 10 – 8
 - recursive calls, 9 – 10
 - redo log space requests, 10 – 9
 - session memory, 8 – 16
 - shared server processes, 10 – 6 to 10 – 8
 - sorts (disk), 11 – 2
 - sorts (memory), 11 – 2
 - undo block, 10 – 2
 - undo header, 10 – 2
- STORAGE clause
 - CREATE TABLE command, 9 – 5
 - examples, 9 – 5
 - modifying SQL.BSQ, 9 – 7
 - OPTIMAL, 9 – 12
 - parallel query option, 6 – 5, 6 – 20
 - parameter values, 9 – 7
- stored procedures
 - application uses, 2 – 12
 - BEGIN_DISCRETE_TRANSACTION, 4 – 9
 - KEEP, 4 – 15
 - performance benefits, 2 – 10, 2 – 11
 - READ_MODULE, 3 – 6
 - registering with the database, 3 – 2
 - SET_ACTION, 3 – 4
 - SET_CLIENT_INFO, 3 – 5
 - SET_MODULE, 3 – 3 to 3 – 4
 - SIZES, 4 – 15
 - UNKEEP, 4 – 16
- striping tables, 9 – 5
 - examples, 9 – 5
- subqueries, converting to joins, 5 – 17

- swapping
 - library cache, 8 – 11
 - reducing, 8 – 4
 - SGA, 8 – 24
- System Global Area (SGA), tuning, 8 – 4

T

- tables
 - parallel creation, 6 – 4
 - placement on disk, 9 – 6
 - STORAGE clause with parallel query
 - option, 6 – 5
 - striping examples, 9 – 5
 - summary or rollup, 6 – 4
- TABLESPACE clause
 - CREATE TABLE command, 9 – 5 to 9 – 7
 - examples, 9 – 5 to 9 – 7
- throughput, 5 – 7
 - cost-based approach, 5 – 27 to 5 – 29
 - optimizing, 7 – 12 to 7 – 13, 7 – 16
- TIMED_STATISTICS, A – 8
- TIMESTAMP column,
 - PLAN_TABLE table, A – 23
- TKPROF program, A – 7, A – 10
 - command line parameters, A – 11
 - editing the output SQL script, A – 19
 - example of output, A – 15
 - generating the output SQL script, A – 18
 - syntax, A – 10
 - using the EXPLAIN PLAN command,
 - A – 11
- TKPROF_TABLE
 - columns of, A – 20
 - querying, A – 19
- TOTALQ column,
 - V\$QUEUE table, 10 – 5, 10 – 6
- trace facility. *See* Oracle Trace; SQL trace facility
- transactions, 4 – 7
 - assigning rollback segments, 9 – 12
 - discrete, 4 – 8
- Trusted Oracle Server, application registration,
 - 3 – 3

- tuning, 1 – 1
 - buffer cache, 8 – 18 to 8 – 24
 - checkpoints, 11 – 8
 - client/server applications, 2 – 10
 - contention, 10 – 1
 - data dictionary cache, 8 – 14
 - decision support systems, 2 – 3 to 2 – 4
 - and design, 1 – 5 to 1 – 6
 - distributed databases, 2 – 5 to 2 – 6
 - goals, 1 – 4 to 1 – 5
 - I/O, 9 – 2
 - identifying bottlenecks, 1 – 7, A – 2
 - library cache, 8 – 9
 - memory allocation, 8 – 2, 8 – 24
 - monitoring registered applications, 3 – 2
 - multi-purpose applications, 2 – 8
 - multi-threaded server, 10 – 4 to 10 – 7
 - OLTP applications, 2 – 2 to 2 – 3
 - operating system, 1 – 6, 8 – 3
 - parallel query option, 2 – 7 to 2 – 8, 6 – 13 to 6 – 20
 - Parallel Server, 2 – 7
 - production systems, 1 – 6 to 1 – 7
 - query servers, 6 – 17, 10 – 8
 - scientific or statistical systems, 2 – 4 to 2 – 5
 - shared pool, 8 – 8 to 8 – 17
 - sorts, 11 – 2
 - SQL and PL/SQL areas, 8 – 6
 - SQL statements, 7 – 1
 - System Global Area (SGA), 8 – 4

U

- undo header statistic, 10 – 2
- undo block statistic, 10 – 2
- unique keys
 - optimization, 5 – 18
 - searches, 5 – 34
- UNIQUENESS column, 7 – 40
- UNKEEP procedure, 4 – 16
- UNRECOVERABLE option, 6 – 20
- USE_CONCAT hint, 7 – 25
- USE_MERGE hint, 7 – 27
- USE_NL hint, 7 – 26
- USER_DUMP_DEST, A – 8

- USER_ID column, TKPROF_TABLE, A – 20
- USER_INDEXES view, 7 – 40
- USER_TAB_COLUMNS view, 5 – 45
- USER_TABLES view, 5 – 45
- users, memory allocation, 8 – 6
- UTLCHAIN.SQL, 9 – 8
- UTLXPLAN.SQL, A – 22

V

- V\$ dynamic performance views, 1 – 8
- V\$DATAFILE view, disk I/O, 9 – 3
- V\$DISPATCHER view, using, 10 – 4
- V\$FILESTAT view
 - disk I/O, 9 – 3
 - PHYRDS column, 9 – 3
 - PHYWRDS column, 9 – 3
- V\$LATCH view
 - GETS column, 10 – 11
 - MISSES column, 10 – 11
 - SLEEPS column, 10 – 11
 - using, 10 – 10
- V\$LIBRARYCACHE view
 - NAMESPACE column, 8 – 9
 - PINS column, 8 – 10
 - RELOADS column, 8 – 10
 - using, 8 – 9
- V\$QUEUE view
 - examining wait times, 10 – 5
 - identifying contention, 10 – 6
- V\$ROWCACHE view
 - data dictionary cache performance statistics, 8 – 15
 - GETMISSES column, 8 – 15, 8 – 16
 - GETS column, 8 – 15 to 8 – 16
 - PARAMETER column, 8 – 15
 - using, 8 – 15
- V\$SESSION, application registration, 3 – 2
- V\$SESSION_WAIT view, 1 – 7, A – 2 to A – 4
- V\$SESSTAT view, using, 8 – 16
- V\$SQLAREA,
 - application registration, 3 – 2, 3 – 5

V\$SYSSTAT view
 detecting dynamic extension, 9 – 10
 examining recursive calls, 9 – 10
 redo buffer space, 10 – 9
 tuning sorts, 11 – 2
 using, 8 – 18
V\$WAITSTAT view
 reducing free list contention, 11 – 7
 rollback segment contention, 10 – 2
views, optimization, 5 – 19 to 5 – 29
virtual tables
 X\$KCBCBH table, 8 – 22
 X\$KCBRBH table, 8 – 19

W
WAIT column, V\$QUEUE table, 10 – 5, 10 – 6
writing SQL statements, 7 – 2

X
X\$KCBCBH table, 8–22
X\$KCBRBH table
 buffer cache performance statistics, 8 – 19
 COUNT column, 8 – 19
 enabling use, 8 – 19
 INDX column, 8 – 19

Reader's Comment Form

Name of Document: Oracle7 Server Tuning
Part No. A32537-1

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the topic, chapter, and page number below:

Please send your comments to:

Server Technologies Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood City, CA 94065, U.S.A.
Fax: (415) 506-7200

If you would like a reply, please give your name, address, and telephone number below:

Thank you for helping us improve our documentation.

1875

1876

1877

1878

1879

1880

1881

1882

1883

1884

1885

1886

1887

1888

1889

1890

1891

1892

1893

1894

1895

1896

1897

1898

1899

1900

1901

1902

1903

1904

1905

1906

1907

1908

1909

1910

1911

1912

1913

1914

1915

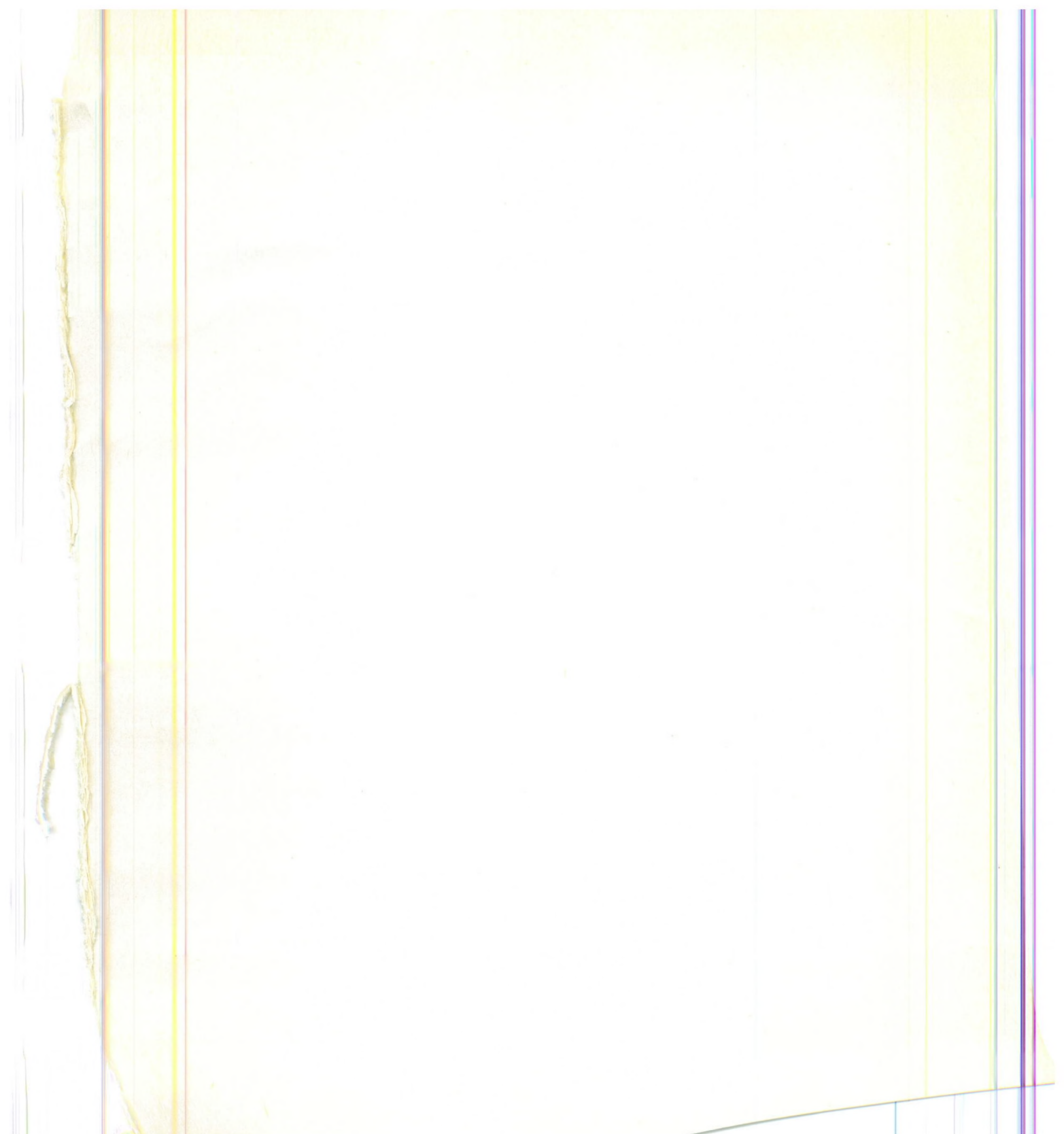
1916

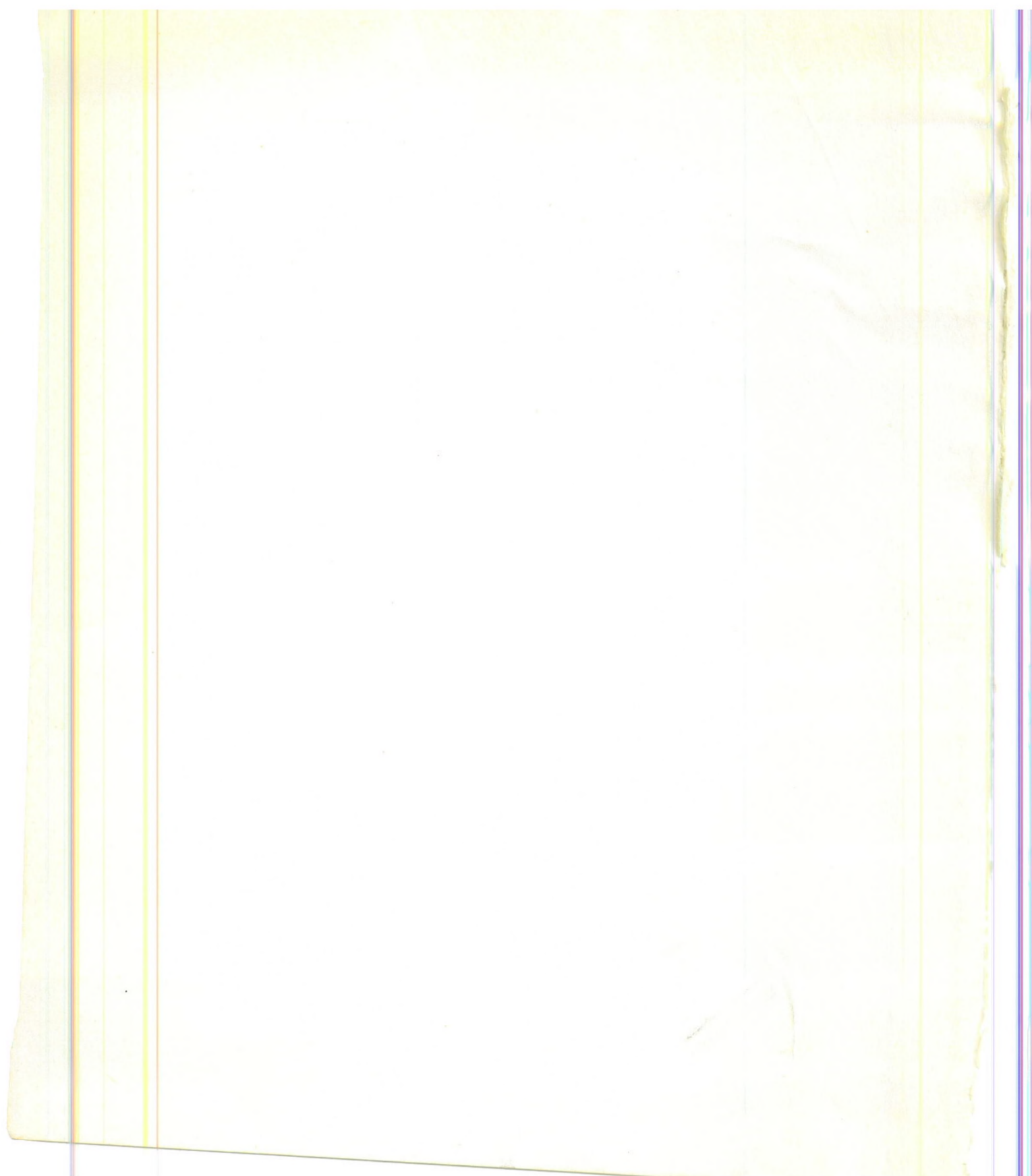
1917

1918

1919

1920







ORACLE®